1-1-2001

The alternating Schwarz method: Mathematical foundation and parallel implementation

Bernd Flemisch
Iowa State University

The alternating Schwarz method:
Mathematical foundation and parallel implementation

by

Bernd Flemisch

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Mathematics
Major Professor: Don Pigozzi

Iowa State University
Ames, Iowa
2001

Graduate College
Iowa State University

This is to certify that the Master's thesis of

Bernd Flemisch

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy

TABLE OF CONTENTS

INTRODUCTION

The alternating Schwarz method belongs to the class of domain decomposition methods for partial differential equations. Domain decomposition methods can be regarded as divide and conquer algorithms. The given domain is partitioned into a number of subregions. The original problem can be reformulated as a family of subproblems of reduced size defined on the subdomains. Based on solving these subproblems, a preconditioner is constructed for the system of linear equations which evolves from the discretization of the original problem. Using this preconditioner, the solution for the linear system is obtained with a preconditioned Krylov subspace method.

One of the major advantages of domain decomposition methods is the natural parallelism in solving the subproblems defined on the subdomains. With the upcoming of parallel computer architectures, domain decomposition methods have become very popular during the last two decades of the twentieth century. However, their origin can be found in the year 1870. In (Sc70), Hermann Amandus Schwarz introduced an algorithm to prove the existence of harmonic functions on irregular shaped domains. Today, this algorithm is known as the alternating Schwarz method. The purpose of this thesis is to discuss the mathematical aspects of the alternating Schwarz method, and to give insight into its implementation on a distributed memory machine.

Chapter 1 provides an introduction to the finite element method. We start with the classical formulation of the Poisson problem and derive its variational formulation. Based on this formulation, the concept of discretization of the problem via triangulation of the domain is presented, leading to the finite element approximation problem and the corresponding system of linear equations. The generalization to the case of a symmetric elliptic differential operator

is mentioned. We also discuss an algorithm used to discretize the problem, and to solve the linear system with the conjugate gradient method.

In chapter 2 we consider the mathematical foundation of the alternating Schwarz method. We begin with the classical formulation of the multiplicative variant of the method and obtain its variational formulation and also give a characterization in terms of projection operators. On the finite dimensional level, we focus on the additive Schwarz method. The construction of the additive Schwarz preconditioner is presented, and the resulting preconditioned conjugate gradient method is introduced.

Chapter 3 is dedicated to a discussion of the parallel implementation of the additive Schwarz method. To become familiar with the concepts of parallel computing, a characterization of a distributed memory machine is given, and the basics of the message passing system MPI are introduced. Then, we see how the discretization of the problem and the process of obtaining the system of linear equations can be carried out in parallel. The preconditioned conjugate gradient method is parallelized, and a compact form of the parallel algorithm is given. Finally, the results of some numerical experiments are presented.

Appendix A reviews the definitions of some function spaces which are often used througout the thesis. The notions of Banach, Hilbert, L^p, and Sobolev spaces are introduced. Appendix B includes the Fortran code which is used for the implementation of the additive Schwarz method.

1
OVERVIEW OF THE FINITE ELEMENT METHOD

The basic idea in any numerical method for approximating the solution of a differential equation is to *discretize* the given continuous problem with infinitely many degrees of freedom to obtain a system of equations with only finitely many unknowns that can be solved using a computer. This chapter introduces the finite element method applied to the example of the Poisson problem. In addition to the finite difference method, the finite element method is one of the most commonly used techniques for approximating the solution of a partial differential equation. Various books are available on this subject, for example (Qu94) and (Jo87).

One should be familiar with the notion of *Hilbert* spaces and *Sobolev* spaces. They will be often used throughout this thesis. Appendix A gives a review of their basic definitions.

1.1 Classical and variational formulation

To introduce the finite element method, we consider the Poisson problem with homogeneous Dirichlet boundary conditions,

$$\begin{aligned} -\Delta u &= f \quad \text{in} \quad \Omega, \\ u &= 0 \quad \text{on} \quad \Gamma, \end{aligned} \tag{1.1}$$

where Ω is an open 2-dimensional domain with Lipschitz boundary Γ, f a given function in $L^2(\Omega)$ and $\Delta = \sum_{j=1}^{2} D_j D_j$ the Laplace operator, where D_j denotes the partial derivative with respect to $x_j, j = 1, 2$. Let $\mathbf{n} = (n_1, n_2)$ be the outward unit normal vector to Γ (see Figure 1.1). We indicate with $\overline{\Omega}$ the closure of the domain Ω, and with $C^2(\overline{\Omega})$ the space of twice continuously differentiable functions in $\overline{\Omega}$. Under the classical asumption that the function f is Hölder continuous, problem (1.1) has a unique solution $u \in C^2(\overline{\Omega})$ satisfying $u = 0$ on

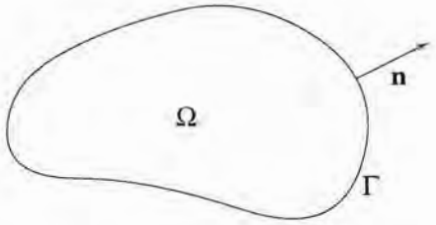

Figure 1.1 The domain Ω.

Γ. This is a well-known fact from the theory of partial differential equations, see for instance (Gi77).

The discretization process starts from a reformulation of the given differential equation as a *variational* or *weak problem.* The well-known *Green's* formula of integration by parts will be of great importance to show the equivalence of both formulations. Let $v, w \in C^2(\overline{\Omega})$, and denote by $d\mathbf{x}$ the element of area in $\mathbf{R}^2$, by ds the element of arc length along Γ and by ∇v the *gradient* of v, i.e., $\nabla v := (D_1 v, D_2 v)$. Then Green's formula reads

$$\int_\Omega \nabla v \nabla w \, d\mathbf{x} = \int_\Gamma v \frac{\partial w}{\partial \mathbf{n}} \, ds - \int_\Omega v \Delta w \, d\mathbf{x}, \tag{1.2}$$

where

$$\frac{\partial w}{\partial \mathbf{n}} := D_1 w \, n_1 + D_2 w \, n_2,$$

is the *normal derivative*, namely, the derivative in the outward normal direction to the boundary Γ. To give the variational formulation, we need the usual inner product of $L^2(\Omega)$ which is defined by

$$(v, w) := \int_\Omega vw \, d\mathbf{x},$$

and the *bilinear form* $a(\cdot, \cdot)$, given by

$$a(v, w) := (\nabla w, \nabla v).$$

Let us also introduce the *Sobolev* spaces

$$H^1(\Omega) := \{v \in L^2(\Omega) : D_j v \in L^2(\Omega), j = 1, 2\}, \quad \text{and} \tag{1.3}$$

$$V := H_0^1(\Omega) := \{v \in H^1(\Omega) : v_{|\Gamma} = 0\}, \tag{1.4}$$

where $v_{|\Gamma}$ denotes the trace of v on Γ. Now we can state the variational formulation of problem (1.1). Find $u \in V$ such that

$$a(u, v) = (f, v) \quad \forall v \in V. \tag{1.5}$$

The existence and uniqueness of a solution to (1.5) follows from the *Lax-Milgram lemma*, which is a consequence of the well-known Riesz representation theorem for Hilbert spaces. It reads as follows, a complete proof can be found for instance in (Gi77).

THEOREM 1.1 (LAX-MILGRAM LEMMA) *Let V be a Hilbert space, endowed with the inner product $(\cdot, \cdot)_V$ and the norm $||\cdot||_V$. Assume that $\mathcal{F} : V \to \mathbf{R}$ is a linear continuous functional and that $\mathcal{A} : V \times V \to \mathbf{R}$ is a bilinear form such that $\mathcal{A}$ is continuous, namely,*

$$\exists \gamma > 0 : |\mathcal{A}(v, w)| \leq \gamma ||v||_V ||w||_V \quad \forall v, w \in V,$$

and coercive, namely,

$$\exists \alpha > 0 : \mathcal{A}(v, v) \geq \alpha ||v||_V^2 \quad \forall v \in V.$$

Then there exists a unique solution $u \in V$ to

$$\mathcal{A}(u, v) = \mathcal{F}(v) \quad \forall v \in V.$$

There are well established facts that $V = H_0^1(\Omega)$ is a Hilbert space, that $\mathcal{F}(\cdot) := (f, \cdot)$ is a linear continuous functional, and that $a(\cdot, \cdot)$ is continuous. It follows from the *Poincaré* inequality that $a(\cdot, \cdot)$ is also coercive. The Poincaré inequality states that there exists a constant $C_\Omega > 0$ such that

$$\|v\|_{L^2(\Omega)}^2 \leq C_\Omega \|\nabla v\|_{L^2(\Omega)}^2 \qquad \forall v \in V. \tag{1.6}$$

Therefore, taking an arbitrary $\theta \in (0, 1)$,

$$\begin{aligned} a(v, v) = \|\nabla v\|_{L^2(\Omega)}^2 &= \theta \|\nabla v\|_{L^2(\Omega)}^2 + (1 - \theta) \|\nabla v\|_{L^2(\Omega)}^2 \\ &\geq \frac{\theta}{C_\Omega} \|v\|_{L^2(\Omega)}^2 + (1 - \theta) \|\nabla v\|_{L^2(\Omega)}^2 \\ &\geq \min \left\{ \frac{\theta}{C_\Omega}, (1 - \theta) \right\} \|v\|_{1,\Omega}^2 \qquad \forall v \in V. \end{aligned}$$

We can apply the Lax-Milgram lemma and get the desired unique solution u.

We will show that a solution u of the classical problem (1.1) satisfies (1.5). Note that u is an element of V. Indicating with $C_0^\infty(\Omega)$ the space of infinitely often continuously differentiable functions with compact support in Ω, we multiply $(1.1)_1$ with an arbitrary $v \in C_0^\infty(\Omega)$ and integrate over Ω. Applying formula (1.2), we get

$$(f, v) = -\int_\Omega \Delta u v \, d\mathbf{x} = -\int_\Gamma \frac{\partial u}{\partial \mathbf{n}} v \, ds + \int_\Omega \nabla u \nabla v \, d\mathbf{x} = a(u, v),$$

where the integral over Γ vanishes since $v \in C_0^\infty(\Omega)$. Note that C_0^∞ is dense in $H_0^1(\Omega)$. For any $v \in V = H_0^1(\Omega)$ we can find a sequence $\{v_n\}_{n=1}^\infty$ with functions $v_n \in C_0^\infty(\Omega)$ which converges to v with respect to the norm $||\cdot||_{1,\Omega}$. The continuity of the inner product $(\cdot, \cdot)$ and the bilinear form $a(\cdot, \cdot)$ then implies $(f, v) = a(u, v)$ for each $v \in V$.

Now let u satisfy (1.5) and assume that u is sufficiently regular, namely, $u \in C^2(\overline{\Omega})$, and $u = 0$ on Γ. Note that the boundary condition $(1.1)_2$ is satisfied. Taking an arbitrary nonzero $v \in C^2(\overline{\Omega})$ satisfying $v = 0$ on Γ, and using formula (1.2) we have

$$\begin{aligned} 0 = (f, v) - a(u, v) &= \int_\Omega f v \, d\mathbf{x} - \int_\Omega \nabla u \nabla v \, d\mathbf{x} \\ &= \int_\Omega f v \, d\mathbf{x} - \int_\Gamma v \frac{\partial u}{\partial \mathbf{n}} \, ds + \int_\Omega v \Delta u \, d\mathbf{x} \\ &= \int_\Omega v(f + \Delta u) \, d\mathbf{x}, \end{aligned}$$

where the boundary integral vanishes since $v = 0$ on Γ. Since v is arbitrary, it follows that $-\Delta u = f$ on Ω.

1.2 Finite element approximation

The next step is the triangulation of Ω. In the sequel, assume that $\Omega \subset \mathbf{R}^2$ is a *polygonal* domain, i.e., that Γ is a polygon. If in fact Γ was curved, an intermediate step would be to approximate Γ with a polygonal curve, see for example (Jo87, ch.12). The finite element approximation is based on a finite triangulation

$$\overline{\Omega} = \bigcup_{K \in \mathcal{T}^h} K$$

where

- $\mathcal{T}^h$ is a collection of triangles K with a non-empty interior $Int(K)$,
- $Int(K_1) \cap Int(K_2) = \emptyset$ for each distinct $K_1, K_2 \in \mathcal{T}^h$,
- if $F = K_1 \cap K_2 \neq \emptyset$, then F is a common side or vertex of K_1 and K_2,
- $\text{diam}(K) \leq h$ for each $K \in \mathcal{T}^h$.

$\mathcal{T}^h$ is called a *triangulation* of $\overline{\Omega}$, h its *mesh parameter*. Figure 1.2 shows an example.

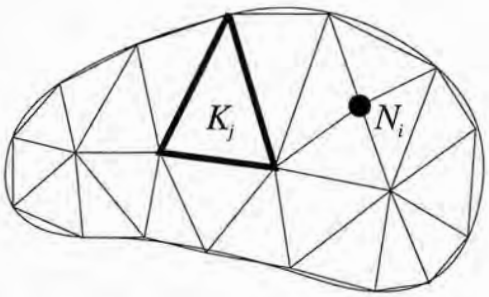

Figure 1.2 Triangulation $\mathcal{T}^h$ of Ω.

Let V^h denote a finite dimensional subspace of $H_0^1(\Omega)$. Usually, V^h is given by piecewise polynomials. We will restrict the dicussion to the case where V^h consists of piecewise linear functions. If we define

$$X^h := \{v_h \in C^0(\overline{\Omega}) : v_{h|K} \text{ is linear } \forall K \in \mathcal{T}^h\},$$

denoting by $C^0(\overline{\Omega})$ the space of continuous functions on $\overline{\Omega}$, we set

$$\begin{aligned} V^h &:= \{v_h \in X^h : v_{h|\Gamma} = 0\} \\ &= X^h \cap H_0^1(\Omega). \end{aligned}$$

A *Galerkin* finite element approximation of the variational problem (1.5) is defined as follows: Find $u_h \in V^h$ such that

$$a(u_h, v_h) = (f, v_h) \quad \forall v_h \in V^h \tag{1.7}$$

The existence of a unique solution follows again from the Lax-Milgram lemma (1.1). The error $||u - u_h||_{1,\Omega}$ to the solution of (1.5) depends on the choice of h and the triangulation $\mathcal{T}^h$. It

can be shown (see for example (Qu99, sec.2.1)) that

$$||u - u_h||_{1,\Omega} \leq Ch|u|_{2,\Omega}, \tag{1.8}$$

provided that $u \in H^2(\Omega)$. The constant C is independent of h. Hence, if the exact solution u is sufficiently regular, the error will go to zero as h goes to zero.

In order to find a solution to problem (1.7), we derive an equivalent system of linear equations. The finite element *nodes* N_i, $i = 1, \ldots, n_h$, of the triangulation $\mathcal{T}^h$ are the vertices of the triangles $K \in \mathcal{T}^h$. We exclude the nodes on the boundary since $v_h = 0$ on Γ. As parameters to describe a function $v_h \in V^h$ we take the values $v_h(N_i)$ at the nodes N_i, $i = 1, \ldots, n_h$. The corresponding basis functions φ_j in V^h are defined by

$$\varphi_j(N_i) := \delta_{ij} = \begin{cases} 1 & : \quad i = j \\ 0 & : \quad i \neq j \end{cases} \qquad i, j = 1, \ldots, n_h.$$

Note that the support of φ_j consists of the triangles with the common vertex N_j, see Figure 1.3. The function v_h can be represented through

$$v_h(\mathbf{x}) = \sum_{j=1}^{n_h} v_h(N_j)\varphi_j(\mathbf{x}). \tag{1.9}$$

We form the column vectors

$$\mathbf{u} := (u_h(N_j))_{j=1,\ldots,n_h},$$

$$\mathbf{f} := ((f, \varphi_j))_{j=1,\ldots,n_h},$$

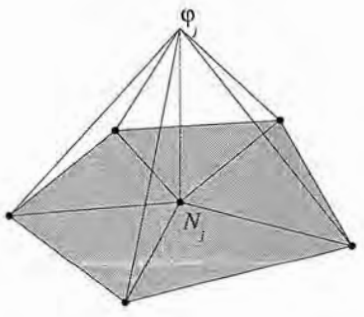

Figure 1.3 The support of φ_j.

and the finite element *stiffness matrix* $A = (a_{ij})_{i,j=1,\ldots,n_h}$ given by

$$a_{ij} := a(\varphi_i, \varphi_j), \quad i,j = 1, \ldots, n_h.$$

In the sequel, we show that this leads to the equivalent formulation of the finite element approximation problem (1.7),

$$A\mathbf{u} = \mathbf{f}. \tag{1.10}$$

Since the bilinear form $a(\cdot,\cdot)$ is symmetric and coercive, the matrix A is symmetric and positive definite. Therefore, A is nonsingular and system (1.10) has a unique solution vector $\mathbf{u}$. To show the equivalence between (1.7) and (1.10), we use the representation of a function $u_h \in V^h$ by (1.9) and the linearity of the inner product $(\cdot,\cdot)$ and the bilinear form $a(\cdot,\cdot)$.

Let $u_h \in V^h$ satisfy (1.7). Then $(A\mathbf{u})_i$, the i-th component of the matrix-vector product $A\mathbf{u}$, is equal to $\mathbf{f}_i$ for all $i = 1, \ldots, n_h$:

$$\begin{aligned}(A\mathbf{u})_i &= \sum_{j=1}^{n_h} a(\varphi_i, \varphi_j) u_h(N_j) = a\left(\varphi_i, \sum_{j=1}^{n_h} u_h(N_j)\varphi_j\right) \\ &= a(\varphi_i, u_h) \\ &= (f, \varphi_i) = \mathbf{f}_i \qquad \forall i = 1, \ldots, n_h\end{aligned}$$

To get the other implication, assume $\mathbf{u}$ satisfies (1.10). Let $u_h \in V^h$ be the function that has as nodal values $u_h(N_j)$ the j-th component of the vector $\mathbf{u}$. We will show that u_h satisfies the condition (1.7). Let $v_h \in V^h$ be an arbitrary function. Then,

$$\begin{aligned}a(u_h, v_h) &= a\left(\sum_{j=1}^{n_h} u_h(N_j)\varphi_j, \sum_{i=1}^{n_h} v_h(N_i)\varphi_i\right) \\ &= \sum_{i=1}^{n_h} v_h(N_i)\left(\sum_{j=1}^{n_h} a(\varphi_i, \varphi_j) u_h(N_j)\right) \\ &= \sum_{i=1}^{n_h} v_h(N_i)(A\mathbf{u})_i \overset{(1.10)}{=} \sum_{i=1}^{n_h} v_h(N_i)\mathbf{f}_i \\ &= \sum_{i=1}^{n_h} v_h(N_i)(f, \varphi_i) = \left(f, \sum_{i=1}^{n_h} v_h(N_i)\varphi_i\right) \\ &= (f, v_h).\end{aligned}$$

We showed that solving system (1.10) determines an approximate solution of the weak problem (1.5). If the solution of (1.5) is sufficiently regular, we therefore approximate the solution to the classical problem (1.1). Section 1.4 will give more insight into algorithmic details for deriving and solving system (1.10).

1.3 Generalization

We can extend the concepts introduced above to the homogeneous Dirichlet boundary value problem

$$\left.\begin{aligned} \mathcal{L}u &= f \quad \text{in} \quad \Omega, \\ u &= 0 \quad \text{on} \quad \Gamma, \end{aligned}\right\} \tag{1.11}$$

where $\mathcal{L}$ is the *symmetric elliptic* operator

$$\mathcal{L}v := -\sum_{j,l=1}^{2} D_j(c_{jl}D_l v) + c_0 v,$$

with coefficients c_0 and c_{jl} belonging to $L^\infty(\Omega)$ and $c_0(\mathbf{x}) \geq 0$ for almost all $\mathbf{x} \in \Omega$. The operator $\mathcal{L}$ is called elliptic if

$$\sum_{j,l=1}^{2} c_{jl}(\mathbf{x})\xi_j\xi_l \geq \alpha|\xi|^2 \quad \forall \xi \in \mathbf{R}^2, \text{ for almost all } \mathbf{x} \in \Omega, \tag{1.12}$$

for a positive constant α. The operator $\mathcal{L}$ is symmetric if

$$c_{jl}(\mathbf{x}) = c_{lj}(\mathbf{x}) \quad \forall j, l = 1, 2, \text{ for almost all } \mathbf{x} \in \Omega.$$

Note that the Laplace operator $-\Delta$ is symmetric elliptic with constant coefficient functions $c_{11} = c_{22} = 1$, and $c_{12} = c_{21} = c_0 = 0$. The bilinear form associated with the operator $\mathcal{L}$ is given by

$$a(v, w) := \int_\Omega \left(\sum_{j,l=1}^{2} c_{jl} D_j v D_l w + c_0 v w \right) d\mathbf{x}.$$

Again, $a(\cdot, \cdot)$ is symmetric and continuous in $H_0^1(\Omega)$. The Poincaré inequality (1.6), together with (1.12) and the condition $c_0(\mathbf{x}) \geq 0$, implies that $a(\cdot, \cdot)$ is also coercive.

In the same way as above, we can derive the weak formulation,

$$\text{find } u \in H_0^1(\Omega): \quad a(u, v) = (f, v) \quad \forall v \in H_0^1(\Omega), \tag{1.13}$$

and the finite element approximation problem,

$$\text{find } u_h \in V_h : \quad a(u_h, v_h) = (f, v_h) \quad \forall v_h \in V^h. \tag{1.14}$$

The existence of unique solutions to (1.13) and (1.14) follows again from the Lax-Milgram lemma (Theorem 1.1). Furthermore, we get the corresponding system of linear equations

$$A\mathbf{u} = \mathbf{f}, \tag{1.15}$$

where the entries in the element stiffness matrix A are given by $a_{ij} := a(\varphi_i, \varphi_j)$.

1.4 A serial algorithm

An algorithm for approximating the solution of problem (1.11) is composed of two basic steps.

1. The *initialization process*, consisting of
   (a) *generating a mesh* of finite elements over the domain Ω and construction of the finite dimensional space V^h to derive (1.14), and
   (b) *assembly* of the stiffness matrix A and the right hand side $\mathbf{f}$ leading to (1.15).
2. *Solution* of (1.15).

1.4.1 Mesh generation

The solutions obtained by the finite element method are always approximate and if a poor domain discretization is used the results may be significantly different from the true solution. The constant C in the error estimate (1.8) depends on the smallest angle of the triangles $K \in \mathcal{T}^h$. One would like to have nearly equilateral triangles throughout the triangulation. It is also desirable that the triangles are small in the parts of Ω where the exact solution varies rapidly and may be larger elsewhere. So-called *adaptive* methods iteratively refine triangulations where necessary, using the information from an a posteriori error estimator of the finite element solution. A detailed description of these methods may be found for example in (Ji90) and (Pe87).

It is sufficient for our purposes to generate a *quasi-uniform* mesh where the triangles have essentially the same size in all parts of Ω. A simplified version of the algorithm proposed in (Lo85) will be used. It consists of the basic two steps:

- generation of nodes on the boundary and in the interior,
- triangle generation.

In the following the algorithm is described in more detail. Figures 1.4 - 1.6 illustrate the process applied to the example of the unit square with mesh parameter 1/3.

As a polygonal curve the boundary Γ consists of segments of straight lines. New nodes are generated on each boundary segment such that the spacing between two neighbouring nodes is equal to the mesh parameter h. Next, imaginary vertical lines are drawn through Ω such that the distance between two consecutive lines is also equal to h. On these vertical lines new nodes are generated in a similar way to the node generation on the boundary segments (see Figure 1.4).

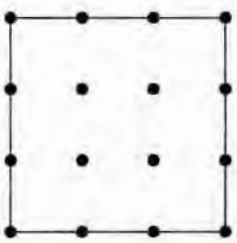

Figure 1.4 The nodes are generated.

For the triangle generation the algorithm uses the technique of an advancing front. At the start of the process the front consists of the boundary segments. The last segment in the front is chosen as a base. Among the available nodes the nearest one to the base is selected to be the apex of the new triangle. Then the front is adjusted by adding the newly formed sides of the generated triangle and eliminating the segments which cannot form any more elements. Figure 1.5 shows an intermediate stage of the process. The thicker line is the advancing front. The segment AB was chosen as base, node C is selected to be the apex of the new element.

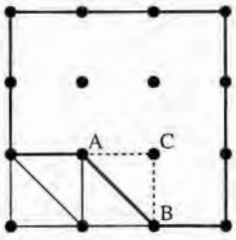

Figure 1.5 ABC will be the new element.

After that, AB will be deleted from the front and the new segments AC and BC will be added to it. The process is repeated until the advancing front consists of the empty set. Then Ω is fully covered by triangles. In Figure 1.6 you can see the complete triangulation.

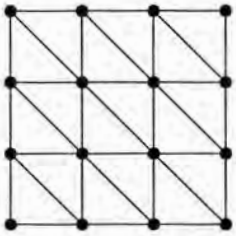

Figure 1.6 The final mesh.

1.4.2 Assembly of the stiffness matrix

The elements of the stiffness matrix A and the right hand side $\mathbf{f}$ are computed by summing the contributions from the different triangles:

$$a_{ij} = a(\varphi_i, \varphi_j) = \sum_{K \in T_h} a_K(\varphi_i, \varphi_j),$$

$$\mathbf{f}_i = \sum_{K \in T_h} (f, \varphi_i)_K,$$

where, in case of Poisson's equation,

$$a_K(\varphi_i, \varphi_j) := \int_K \nabla\varphi_i \nabla\varphi_j,$$

$$(f, \varphi_i)_K := \int_K f\varphi_i$$

As mentioned earlier, the support of φ_j consists of the triangles with the common vertex N_j. Therefore, the only triangles that contribute a nonzero term to a_{ij} are those which have N_i and N_j as vertices. Similarly, only triangles having N_i as one vertex add a nonzero term to $\mathbf{f}_i$. If N_i, N_j and N_k are the vertices of the triangle $K \in \mathcal{T}^h$ then the 3×3-matrix

$$\begin{pmatrix} a_K(\varphi_i, \varphi_i) & a_K(\varphi_i, \varphi_j) & a_K(\varphi_i, \varphi_k) \\ a_K(\varphi_j, \varphi_i) & a_K(\varphi_j, \varphi_j) & a_K(\varphi_j, \varphi_k) \\ a_K(\varphi_k, \varphi_i) & a_K(\varphi_k, \varphi_j) & a_K(\varphi_k, \varphi_k) \end{pmatrix}$$

is called the *element stiffness matrix* for K. The global stiffness matrix A is computed by first calculating the element stiffness matrices for each $K \in \mathcal{T}^h$ and then summing the contributions from each triangle. In a corresponding way the right hand side $\mathbf{f}$ is computed.

1.4.3 Solution of the linear system

As mentioned earlier, system (1.15) has a unique solution vector $\mathbf{u}$. One may choose any kind of direct or iterative algorithm to solve the system. Note that A is an $n_h \times n_h$ matrix. The matrix A is also *sparse*, namely, most of the entries are zero. The support of a basis function φ_j consists of the triangles with common node N_j, and hence, $a_{ij} = 0$ unless N_i and N_j are nodes of the same triangle. Therefore, especially if the system becomes large, an iterative scheme is more appropriate. An intensive discussion of these methods is found for example in (Sa96).

Due to the fact that A is symmetric and positive definite, one can apply the *conjugate gradient* (CG) method. The CG method is a variation of the steepest descent method. It converges after at most n_h iterations. The method was originally invented for solving the quadratic problem

$$\text{minimize } \frac{1}{2}(A\mathbf{u}, \mathbf{u}) - (\mathbf{f}, \mathbf{u})$$

which is equivalent to problem (1.15). We follow the description in (Sa96, sec.6.7). The algorithm reads as follows.

ALGORITHM 1.1 (CONJUGATE GRADIENT ALGORITHM)

1. Start with an initial guess $\mathbf{u}^0$, and set

2. $\mathbf{p}^0 := \mathbf{r}^0 := \mathbf{f} - A\mathbf{u}^0$.

For $k \geq 0$ until convergence, calculate

3. $\alpha_k \quad := \frac{(\mathbf{r}^k, \mathbf{r}^k)}{(A\mathbf{p}^k, \mathbf{p}^k)}$,
4. $\mathbf{u}^{k+1} := \mathbf{u}^k + \alpha_k \mathbf{p}^k$,
5. $\mathbf{r}^{k+1} := \mathbf{r}^k - \alpha_k A\mathbf{p}^k$,
6. $\beta_k \quad := \frac{(\mathbf{r}^{k+1}, \mathbf{r}^{k+1})}{(\mathbf{r}^k, \mathbf{r}^k)}$,
7. $\mathbf{p}^{k+1} := \mathbf{r}^{k+1} + \beta_k \mathbf{p}^k$.

The notation $(\cdot, \cdot)$ indicates the usual dotproduct of two vectors. In each iteration step we check for convergence, after calculating the new residual $\mathbf{r}^{k+1}$ in step 5 of the algorithm. In fact, $\mathbf{r}^{k+1} = \mathbf{f} - A\mathbf{u}^{k+1}$. Convergence can be declared for instance when the maximum norm

$$\max_{i=1,\ldots,n_h} \left| \mathbf{r}_i^{k+1} \right|$$

of $\mathbf{r}^{k+1}$ becomes sufficiently small.

2
MATHEMATICAL ASPECTS OF THE ALTERNATING SCHWARZ METHOD

Any domain decomposition method is based on partitioning the domain Ω into subdomains Ω_i, $i = 1, \ldots, M$. There are two main approaches for decomposing Ω. The first is to use overlapping subdomains. This leads to the *alternating Schwarz* method. The solution is obtained iteratively by solving the subdomain problems and updating the values on the subdomain interfaces. The second approach is to partition Ω into nonoverlapping subdomains. Then, the solution is computed using *iterative substructuring* methods. The same technical tools can be used for the analysis of both approaches. Early efforts to unify the theory behind the two approaches can be found for instance in (Bj89) and (Ch92). We will focus on the case of overlapping subdomains. The classical and variational formulation of the alternating Schwarz method will be introduced. On the finite dimensional level, we will mainly deal with one particular variant, namely, the *additive* Schwarz method. The discussion is restricted to homogeneous Dirichlet boundary conditions. A generalization to other boundary conditions could be made easily but would distract us from focusing on the basic ideas. We follow the description in (Qu99).

2.1 Classical formulation

The alternating Schwarz method was introduced by Hermann Amandus Schwarz as early as 1870 in (Sc70). He used it to prove the existence of harmonic functions on irregular shaped

domains. Let us recall problem (1.11),

$$\left.\begin{aligned} \mathcal{L}u &= f \quad \text{in} \quad \Omega, \\ u &= 0 \quad \text{on} \quad \Gamma. \end{aligned}\right\} \tag{2.1}$$

We decompose Ω into two overlapping subdomains Ω_1 and Ω_2 such that $\Omega = \Omega_1 \cup \Omega_2$. Let $\Gamma_1 := \partial\Omega_1 \cap \Omega_2$ and $\Gamma_2 := \partial\Omega_2 \cap \Omega_1$ denote the artificial boundaries of Ω_1 and Ω_2 in Ω, respectively. Figure 2.1 shows the image with which Schwarz illustrated his method.

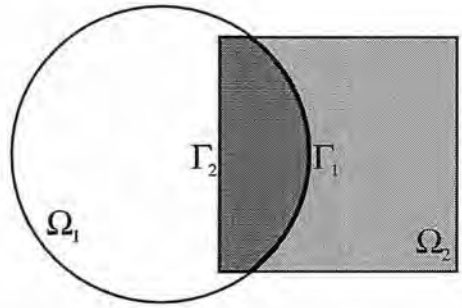

Figure 2.1 Overlapping decomposition of Ω into two subdomains.

If the values of the solution u on Γ_1 and Γ_2 were known, problem (2.1) could be solved by solving the corresponding subproblems on Ω_1 and Ω_2 independently. This gives raise to the formulation of the alternating Schwarz method. It is composed of solving iteratively the following problems.

Let u^0 be an initialization function defined in Ω and vanishing on Γ. For $k \geq 0$ we define two sequences $u^{k+1/2}$ and u^{k+1} by solving respectively

$$\left.\begin{aligned} \mathcal{L}u^{k+1/2} &= f \quad \text{in } \Omega_1, \\ u^{k+1/2} &= u^k \quad \text{on } \Gamma_1, \\ u^{k+1/2} &= 0 \quad \text{on } \partial\Omega_1 \cap \Gamma, \end{aligned}\right\} \tag{2.2}$$

and

$$\left.\begin{aligned} \mathcal{L}u^{k+1} &= f \quad \text{in } \Omega_2, \\ u^{k+1} &= u^{k+1/2} \quad \text{on } \Gamma_2, \\ u^{k+1} &= 0 \quad \text{on } \partial\Omega_2 \cap \Gamma. \end{aligned}\right\} \tag{2.3}$$

The problems (2.2) and (2.3) formulate the *multiplicative* Schwarz method. So far, the functions $u^{k+1/2}$ and u^{k+1} are only defined in $\overline{\Omega_1}$ and $\overline{\Omega_2}$, respectively. Assuming that they are continuous, we extend them to continuous functions defined in $\overline{\Omega}$ by

$$u^{k+1/2} := u^k \quad \text{in } \overline{\Omega} \setminus \overline{\Omega_1}, \text{ and} \tag{2.4}$$

$$u^{k+1} := u^{k+1/2} \quad \text{in } \overline{\Omega} \setminus \overline{\Omega_2}. \tag{2.5}$$

It can be shown that both sequences $u^{k+1/2}$ and u^{k+1} converge to the solution u in the L^∞ norm, as $k \to \infty$. Precisely, there exist constants $C_1, C_2 \in (0,1)$ depending on the structure of (Ω_1, Γ_2) and (Ω_2, Γ_1), respectively, such that for all $k \geq 0$

$$\sup_{\overline{\Omega_1}} |u - u^{k+1/2}| \leq C_1^k C_2^k \sup_{\Gamma_1} |u - u^0|, \tag{2.6}$$

$$\sup_{\overline{\Omega_2}} |u - u^k| \leq C_1^k C_2^{k-1} \sup_{\Gamma_2} |u - u^0|. \tag{2.7}$$

A proof is given for instance in (Li89). Developing the variational formulation, we will see that under certain restrictions the sequences $u^{k+1/2}$ and u^{k+1} converge to the solution u at a geometric rate, in the norm associated with the bilinear form $a(\cdot,\cdot)$.

2.2 Variational formulation

To give the variational formulation of (2.2) and (2.3), we introduce some more notation. Set $V := H_0^1(\Omega)$ as in the previous chapter, and $V_i := H_0^1(\Omega_i)$, $i = 1, 2$. We note that, since $\Omega_i \subset \Omega$, any element in V_i can be extended by zero to an element in V, and that the immersion map from V_i into V is linear and continuous. Therefore, V_1 and V_2 are regarded as subspaces of V. We also need the inner products

$$(v, w)_{\Omega_i} := \int_{\Omega_i} vw,$$

and the bilinear forms

$$a_i(v_i, w_i) := \int_{\Omega_i} \left(\sum_{j,l=1}^{2} c_{jl} D_j v D_l w + c_0 v w \right),$$

on each subdomain Ω_i, $i = 1, 2$.

Since $u^k = 0$ on $\partial\Omega_1 \cap \Gamma$, and $u^{k+1/2} = 0$ on $\partial\Omega_2 \cap \Gamma$, we can rewrite the multiplicative Schwarz method (2.2), and (2.3) as

$$\begin{aligned} \mathcal{L}(u^{k+1/2} - u^k) &= f - \mathcal{L}u^k \quad \text{in } \Omega_1, \\ u^{k+1/2} - u^k &= 0 \qquad\qquad \text{on } \Gamma_1, \\ u^{k+1/2} - u^k &= 0 \qquad\qquad \text{on } \partial\Omega_1 \cap \Gamma, \end{aligned}$$

and

$$\begin{aligned} \mathcal{L}(u^{k+1} - u^{k+1/2}) &= f - \mathcal{L}u^{k+1/2} \quad \text{in } \Omega_2, \\ u^{k+1} - u^{k+1/2} &= 0 \qquad\qquad \text{on } \Gamma_2, \\ u^{k+1} - u^{k+1/2} &= 0 \qquad\qquad \text{on } \partial\Omega_2 \cap \Gamma, \end{aligned}$$

respectively. Introducing $w^{k+1/2} := (u^{k+1/2} - u^k)|_{\Omega_1}$, and $w^{k+1} := (u^{k+1} - u^{k+1/2})|_{\Omega_2}$, the variational formulation of the multiplicative Schwarz method can be given: Start with an initial guess $u^0 \in V$, and for each $k \geq 0$,

$$\left.\begin{aligned} &\text{find } w^{k+1/2} \in V_1 : \quad a_1(w^{k+1/2}, v_1) = (f, v_1)_{\Omega_1} - a_1(u^k, v_1) \qquad \forall v_1 \in V_1, \\ &\text{set } u^{k+1/2} = u^k + w^{k+1/2}, \\ &\text{find } w^{k+1} \in V_2 : \quad a_2(w^{k+1}, v_2) = (f, v_2)_{\Omega_2} - a_2(u^{k+1/2}, v_2) \qquad \forall v_2 \in V_2, \\ &\text{set } u^{k+1} = u^{k+1/2} + w^{k+1}. \end{aligned}\right\} \tag{2.8}$$

We will rewrite the multiplicative Schwarz method in terms of orthogonal projections of V onto the subspaces V_i. This will be very useful for the convergence analysis and the finite element approximation. Let $v_1 \in V_1$. Using (2.8), the weak formulation (1.13), and $V_1 \subset V$, we see that

$$\begin{aligned} a(w^{k+1/2}, v_1) &= a_1(w^{k+1/2}, v_1) \\ &= (f, v_1)_{\Omega_1} - a_1(u^k, v_1) \\ &= (f, v_1) - a(u^k, v_1) = a(u, v_1) - a(u^k, v_1) \\ &= a(u - u^k, v_1). \end{aligned} \tag{2.9}$$

Similarly, it follows that

$$a(w^{k+1}, v_2) = a(u - u^{k+1/2}, v_2) \qquad \forall v_2 \in V_2. \tag{2.10}$$

Let $\mathcal{P}_i$, $i = 1, 2$, be the orthogonal projection of V onto V_i with respect to the inner product induced by the bilinear form $a(\cdot, \cdot)$. Namely, for any $v \in V$ it holds that $\mathcal{P}_i v \in V_i$ satisfies

$$a(\mathcal{P}_i v, w_i) = a(v, w_i) \qquad \forall w_i \in V_i.$$

The relations (2.9), (2.10) can be expressed as

$$w^{k+1/2} = \mathcal{P}_1(u - u^k), \quad w^{k+1} = \mathcal{P}_2(u - u^{k+1/2}),$$

respectively. We see that the corrections $w^{k+1/2}$ and w^{k+1} are the projections of the errors $u - u^k$ and $u - u^{k+1/2}$ onto the subspaces V_1 and V_2, respectively.

The multiplicative Schwarz method (2.8) in terms of the projection operators $\mathcal{P}_i$ reads

$$\begin{aligned} u^{k+1/2} &= u^k + \mathcal{P}_1(u - u^k), \\ u^{k+1} &= u^{k+1/2} + \mathcal{P}_2(u - u^{k+1/2}). \end{aligned} \tag{2.11}$$

Introducing

$$\mathcal{Q}_m := \mathcal{P}_1 + \mathcal{P}_2 - \mathcal{P}_2\mathcal{P}_1,$$

we write (2.11) as a one-step method,

$$\begin{aligned} u^{k+1} &= u^k + \mathcal{P}_1(u - u_k) + \mathcal{P}_2(u - u^k - \mathcal{P}_1(u - u^k)) \\ &= u^k + \mathcal{Q}_m(u - u^k). \end{aligned} \tag{2.12}$$

Let $\mathcal{I}$ denote the identity operator, and define the error $e^k := u - u^k$. From (2.12), we obtain a recursion formula for the error,

$$\begin{aligned} e^{k+1} &= u - (u^k + \mathcal{Q}_m(u - u^k)) \\ &= (\mathcal{I} - \mathcal{Q}_m)(u - u^k) \\ &= (\mathcal{I} - \mathcal{P}_2)(\mathcal{I} - \mathcal{P}_1)e^k. \end{aligned} \tag{2.13}$$

P.J. Lions proved in (Li88) that the operator $(\mathcal{I} - \mathcal{P}_2)(\mathcal{I} - \mathcal{P}_1)$ is a contraction with respect to the norm $\|\cdot\|_a$ associated with the bilinear form $a(\cdot, \cdot)$, namely, there exists a constant $K_0 \in (0, 1)$ such that

$$\|(\mathcal{I} - \mathcal{P}_2)(\mathcal{I} - \mathcal{P}_1)v\|_a \leq K_0\|v\|_a \qquad \forall v \in V.$$

This result requires the condition $V = V_1 + V_2$, i.e., every element $v \in V$ can be written as the sum of an element $v_1 \in V_1$ and an element $v_2 \in V_2$. This asumption is satisfied for a large class of subdomains Ω_1 and Ω_2. As a consequence, the sequence u^{k+1} converges to the solution u at a geometric rate,

$$\|u - u^{k+1}\|_a \leq K_0^{k+1} \|u - u^0\|_a \qquad \forall k \geq 0.$$

Let $\mathcal{G}$ be the solution operator associated with problem (1.11), namely, $\mathcal{G} : L^2(\Omega) \to V$, $\mathcal{G}f := u$. Since the Lax-Milgram lemma (Theorem 1.1) guarantees the existence of a unique solution $u \in V$ for each function $f \in L^2(\Omega)$, the operator $\mathcal{G}$ is well defined. It is easy to see that $\mathcal{G}$ is linear, and the fact that the bilinear form $a(\cdot, \cdot)$ is coercive, together with the Cauchy-Schwarz inequality, implies that $\mathcal{G}$ is also continuous. Using the formal identity $\mathcal{G}\mathcal{L}u^k = u^k$, we write (2.12) as

$$\begin{aligned} u^{k+1} &= u^k + \mathcal{Q}_m(\mathcal{G}f - \mathcal{G}\mathcal{L}u^k) \\ &= u^k + \mathcal{Q}_m\mathcal{G}(f - \mathcal{L}u^k) \end{aligned} \tag{2.14}$$

On the finite dimensional level, the operators in (2.14) will be replaced by their matrix representations.

2.3 Finite element approximation

The multiplicative Schwarz method is adapted to the finite element approximation problem (1.14),

$$\text{find } u_h \in V_h : \quad a(u_h, v_h) = (f, v_h) \quad \forall v_h \in V^h. \tag{2.15}$$

We assume that the artificial boundaries Γ_1 and Γ_2 do not cut any triangle $K \in \mathcal{T}^h$. This implies that the global triangulation $\mathcal{T}^h$ of $\overline{\Omega}$ induces two triangulations $\mathcal{T}_1^h$ of $\overline{\Omega_1}$ and $\mathcal{T}_2^h$ of $\overline{\Omega_2}$ that match in the overlapping region (see Figure 2.2). Let us introduce the finite dimensional subspaces V_i^h of V_i, $i = 1, 2$. Set

$$V_i^h := V_i \cap V^h = \left\{ v_{h|\Omega_i} : v_h \in V^h, v_{h|\Gamma_i} = 0 \right\}, \quad i = 1, 2.$$

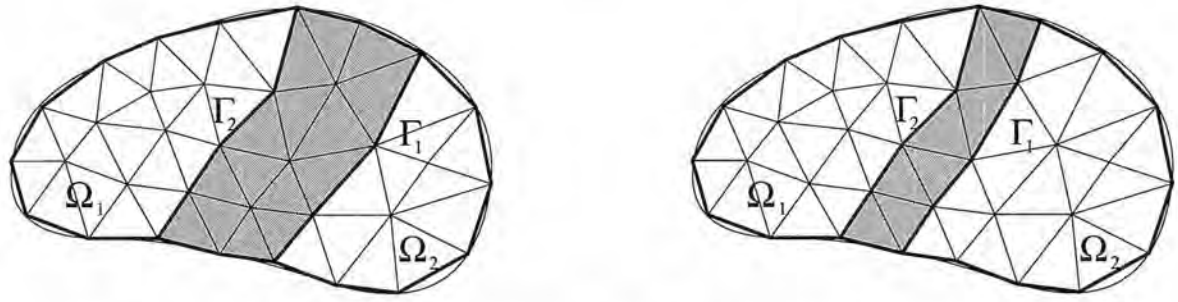

Figure 2.2 Overlapping decomposition of Ω with different layers of overlap.

Similarly to $V_i \subset V$, the spaces V_i^h are considered to be supspaces of V^h. Following (2.8), the multiplicative Schwarz method at the discrete level reads as follows. Start with an initial guess $u_h^0 \in V^h$ and for each $k \geq 0$

$$\begin{aligned}
&\text{find } w_h^{k+1/2} \in V_1^h: \quad a_1(w_h^{k+1/2}, v_{1,h}) = (f, v_{1,h})_{\Omega_1} - a_1(u^k, v_{1,h}) \qquad \forall v_{1,h} \in V_1^h, \\
&\text{set } u_h^{k+1/2} = u_h^k + w_h^{k+1/2}, \\
&\text{find } w_h^{k+1} \in V_2: \quad a_2(w_h^{k+1}, v_{2,h}) = (f, v_{2,h})_{\Omega_2} - a_2(u_h^{k+1/2}, v_{2,h}) \qquad \forall v_{2,h} \in V_2^h, \\
&\text{set } u_h^{k+1} = u_h^{k+1/2} + w_h^{k+1}.
\end{aligned} \tag{2.16}$$

We want to reformulate the method in algebraic terms and derive the discrete analogue of (2.14). Let us recall the system (1.15).

$$A\mathbf{u} = \mathbf{f}, \tag{2.17}$$

where A is the stiffness matrix with elements $a_{ij} = a(\varphi_i, \varphi_j)$. We denote by n_h the number of internal nodes of Ω and by $I := \{1, \ldots, n_h\}$ the set of indices of these nodes. Let I_1 and I_2 be the set of indices of the nodes belonging to the interior of Ω_1 and Ω_2, and let n_1 and n_2 indicate the number of nodes in the interior of Ω_1 and Ω_2, respectively. Due to the overlap of the subdomains Ω_1 and Ω_2, it is possible that $I_1 \cap I_2 \neq \emptyset$ and $n_1 + n_2 > n_h$. The left image in Figure 2.2 shows such an example. Ω is partitioned into two subdomains with two layers of overlap. The shaded area indicates the overlapping region. If we have a decomposition with only one layer of overlap as in the right image, it holds that $I_1 \cap I_2 = \emptyset$ and $n_1 + n_2 = n_h$.

Let us order the indices in such a way that those corresponding to the nodes belonging exclusively to the interior of Ω_1 come first, followed by those corresponding to the nodes

internal to the overlapping region $\Omega_1 \cap \Omega_2$, and finally the ones which correspond to the nodes belonging exclusively to the interior of Ω_2. Let A_{11} and A_{22} denote the submatrices of the stiffness matrix A formed by the first n_1 rows and n_1 columns, and by the last n_2 rows and n_2 columns, respectively (see Figure 2.3). Then A_{11} is the stiffness matrix for the subdomain Ω_1, and A_{22} that of Ω_2. We can relate them to the global stiffness matrix A in terms of extension and restriction matrices. In fact,

$$A_{11} = R_1 A R_1^T, \quad A_{22} = R_2 A R_2^T,$$

where R_i^T and R_i, $i = 1, 2$, are extension and restriction matrices, respectively. Precisely, R_1^T is the $n_h \times n_1$ matrix whose first n_1 rows and columns form the identity matrix, and the entries of the last $n_h - n_1$ rows are all 0, whereas R_2^T is the $n_h \times n_2$ matrix whose last n_2 rows and columns form the identity matrix, and the entries of the first $n_h - n_2$ rows are all 0. Therefore, given a vector $\mathbf{v}^i$ of length n_i of nodal values of a function $v_{i,h} \in V_i^h$, the action of R_i^T on $\mathbf{v}^i$ is

$$(R_i^T \mathbf{v}^i)_j = \begin{cases} \mathbf{v}_j^i & \text{for } j \in I_i \\ 0 & \text{for } j \in I \setminus I_i. \end{cases}$$

The transpose R_i of R_i^T is the matrix whose action restricts a vector $\mathbf{v} \in \mathbf{R}^{n_h}$ to a vector of length n_i by preserving the entries with indices belonging to I_i.

We need the matrix representations of the operators involved in (2.14), in order to rewrite the multiplicative Schwarz method in algebraic terms. It is clear that the differential operator $\mathcal{L}$ corresponds to the element stiffness matrix A, and that the solution operator $\mathcal{G}$ corresponds to

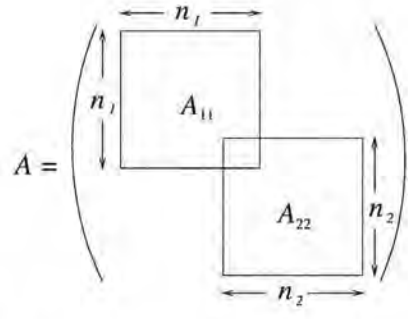

Figure 2.3 The block structure of the stiffness matrix A.

the matrix representation A^{-1}. It remains to find the analogues of the projection operators $\mathcal{P}_i$, $i = 1, 2$. Let φ_j^i, $j = 1, \ldots, n_i$ be the subset of the basis functions with support in Ω_i, and let N_j^i, $j = 1, \ldots, n_i$, indicate the nodes in the interior of Ω_i. Let $u_h \in V^h$ and $w_h := \mathcal{P}_i u_h \in V_i^h$ be represented by $u_h = \sum_{j=1}^{n_h} u_j \varphi_j$ and $w_h = \sum_{j=1}^{n_i} w_j \varphi_j^i$. Note that

$$a(w_h, v_i) = a(u_h, v_i) \qquad \forall v_i \in V_i.$$

Therefore,

$$\sum_{j=1}^{n_i} w_j a(\varphi_j, \varphi_l) = \sum_{j=1}^{n_h} u_j a(\varphi_j, \varphi_l) \qquad \forall l = 1, \ldots, n_i.$$

Denoting by $\mathbf{u}$ and $\mathbf{w}$ the vector of the nodal values of u_h and w_h, respectively, it follows that

$$A_{ii}\mathbf{w} = R_i A \mathbf{u},$$

and therefore,

$$\mathbf{w} = A_{ii}^{-1} R_i A \mathbf{u}.$$

Hence, the matrix representation P_i of the projection operator $\mathcal{P}_i$ is given by

$$P_i := R_i^T A_{ii}^{-1} R_i A.$$

Denoting by

$$B_i := R_i^T A_{ii}^{-1} R_i, \quad i = 1, 2,$$

the discrete analogue of (2.14) is obtained,

$$\begin{aligned} \mathbf{u}^{k+1} &= \mathbf{u}^k + (P_1 + P_2 - P_2 P_1) A^{-1} (\mathbf{f} - A\mathbf{u}^k) \\ &= \mathbf{u}^k + (B_1 + B_2 - B_2 A B_1)(\mathbf{f} - A\mathbf{u}^k). \end{aligned} \tag{2.18}$$

Since the term $B_2 A B_1$ is involved in (2.18), the iteration step cannot be parallelized. If we simply drop this term, the *additive* Schwarz method is obtained,

$$\mathbf{u}^{k+1} = \mathbf{u}^k + (B_1 + B_2)(\mathbf{f} - A\mathbf{u}^k). \tag{2.19}$$

The calculation of the corrections $B_1(\mathbf{f} - A\mathbf{u}^k)$ and $B_2(\mathbf{f} - A\mathbf{u}^k)$ can be carried out concurrently. Due to this higher potential for parallelism, we focus on the implementation of the additive Schwarz method.

It can be shown that the matrix $B_1 + B_2$ is symmetric and positive definite, and, therefore, invertible (see (Qu99, ch.3)). Let us introduce the *additive Schwarz preconditioner*

$$P_{as} := (B_1 + B_2)^{-1}.$$

Note that P_{as} is symmetric and positive definite. The additive Schwarz method can be regarded as a fixed-point iteration method for the system

$$P_{as}^{-1} A\mathbf{u} = P_{as}^{-1}\mathbf{f}. \tag{2.20}$$

The system (2.20) can be derived from (2.17) by taking P_{as} as a preconditioner. Rewriting (2.19) as

$$\mathbf{u}^{k+1} = \mathbf{u}^k + P_{as}^{-1}(\mathbf{f} - A\mathbf{u}^k),$$

we see that each application of the matrix P_{as}^{-1} corresponds to one iteration step of the additive Schwarz method.

As mentioned in (Sm96), the iteration scheme (2.19) is not guaranteed to converge in the general case. But if we use the conjugate gradient method applied to the system (2.20), the iterates $\mathbf{u}^k$ will converge to the solution $\mathbf{u}$. This fact is described in the next section.

REMARK 2.1 So far, we have only discussed the case of two subdomains. The generalization to the case of M subdomains, $M > 2$, is straightforward. Let Ω be partitioned into overlapping subdomains Ω_i, $i = 1, \ldots, M$. The iteration step (2.19) of the additive Schwarz method becomes

$$\mathbf{u}^{k+1} = \mathbf{u}^k + \left(\sum_{i=1}^{M} B_i\right)(\mathbf{f} - A\mathbf{u}^k),$$

where

$$B_i := R_i^T A_{ii}^{-1} R_i, \quad i = 1, \ldots, M.$$

Defining

$$P_{as} := \left(\sum_{i=1}^{M} B_i\right)^{-1}, \tag{2.21}$$

we derive the same preconditioned system (2.20).

2.4 Acceleration with the conjugate gradient method

The preconditioner P_{as} is symmetric and positive definite. Therefore, we can use the conjugate gradient method applied to the preconditioned system (2.20). As described in (Sa96, ch.9), the *preconditioned CG method* is obtained. The iteration scheme reads as follows.

ALGORITHM 2.1 (PRECONDITIONED CONJUGATE GRADIENT ALGORITHM)

1. Assign $\mathbf{u}^0$, and set
2. $\mathbf{r}^0 := \mathbf{f} - A\mathbf{u}^0$,
3. $\mathbf{p}^0 := \mathbf{z}^0 := P_{as}^{-1}\mathbf{r}^0$.

For $k \geq 0$ until convergence, calculate

4. $\alpha_k := \frac{(\mathbf{z}^k, \mathbf{r}^k)}{(\mathbf{p}^k, A\mathbf{p}^k)}$,
5. $\mathbf{u}^{k+1} := \mathbf{u}^k + \alpha_k \mathbf{p}^k$,
6. $\mathbf{r}^{k+1} := \mathbf{r}^k - \alpha_k A\mathbf{p}^k$,
7. $\mathbf{z}^{k+1} := P_{as}^{-1}\mathbf{r}^{k+1}$,
8. $\beta_{k+1} := \frac{(\mathbf{z}^{k+1}, \mathbf{r}^{k+1})}{(\mathbf{z}^k, \mathbf{r}^k)}$,
9. $\mathbf{p}^{k+1} := \mathbf{z}^{k+1} + \beta_{k+1}\mathbf{p}^k$.

The CG method provides us with an estimate for the convergence of the iterates $\mathbf{u}^k$ to the solution vector $\mathbf{u}$ of system (2.17). Let $\kappa(M)$ denote the condition number of a positive definite matrix M, namely,

$$\kappa(M) := \|M\| \, \|M^{-1}\|.$$

Using the 2-norm in the equation above, and denoting by $\lambda_{\max}(M)$ and $\lambda_{\min}(M)$ the largest and smallest eigenvalue of M, respectively, the condition number satisfies

$$\kappa(M) = \frac{\lambda_{\max}}{\lambda_{\min}}.$$

Then, the convergence rate is, as mentioned for instance in (Go96),

$$\|\mathbf{u}^{k+1} - \mathbf{u}\|_A \leq 2 \left(\frac{\sqrt{\kappa(P_{as}^{-1}A)} - 1}{\sqrt{\kappa(P_{as}^{-1}A)} + 1} \right)^{k+1} \|\mathbf{u}^0 - \mathbf{u}\|_A, \tag{2.22}$$

where $\|\mathbf{v}\|_A := \sqrt{(\mathbf{v}, \mathbf{v})_A}$ is the norm associated with the *A-inner product* $(\cdot, \cdot)_A$, given by

$$(\mathbf{v}, \mathbf{w})_A := (A\mathbf{v}, \mathbf{w}) \qquad \forall \mathbf{v}, \mathbf{w} \in \mathbf{R}^{n_h}.$$

Note that the term

$$\left(\frac{\sqrt{\kappa(P_{as}^{-1}A)} - 1}{\sqrt{\kappa(P_{as}^{-1}A)} + 1} \right)$$

is positive and stricly less than 1. The factor 2 in (2.22) shows that we may not achieve convergence at a geometric rate, unlike we saw for the multiplicative Schwarz method.

We denote by H the maximal width of the subdomains Ω_i, and by δ the minimal width of the overlapping regions. It can be shown that the condition number $\kappa(P_{as}^{-1}A)$ is on the order of $1/H\delta$, see (Dr94b). This suggests that the number of iterations needed to reduce the initial error by a fixed percentage will decrease if we increase the overlap. More iterations should be necessary when the number of subdomains M becomes larger and, hence, H becomes smaller. We will see these effects as results of the numerical experiments in the next chapter.

REMARK 2.2 For an efficient implementation of Algorithm 2.1, it is often advantageous to use inexact solvers on the subdomains. This can be done by approximating the local stiffness matrices A_{ii} with an incomplete factorization $\mathcal{A}_{ii}$. The preconditioner P_{as} changes to

$$\widetilde{P_{as}} := \left(\sum_{i=1}^{M} R_i^T \mathcal{A}_{ii}^{-1} R_i \right)^{-1}$$

Since only a different preconditioner is used, but the global stiffness matrix A stays the same, Algorithm 2.1 will still converge to the correct solution.

2.5 The two-level additive Schwarz method

The convergence behaviour of the additive Schwarz method can be improved by adding a coarse grid correction to the preconditoner P_{as} defined in (2.21). We give a brief overview of the concept. A detailed discussion can be found in (Sm96).

Let $\mathcal{T}^H$ be a coarse triangulation of the domain Ω, and let $\mathcal{T}^h$ be a fine triangulation resulting from a refinement of $\mathcal{T}^H$. This is illustrated in Figure 2.4, where the thicker lines form the triangles of $\mathcal{T}^H$. Denoting by V^H the space of continuous, piecewise linear finite element functions on the coarse triangulation $\mathcal{T}^H$, a coarse finite element approximation problem is given by

$$\text{find } u_H \in V^H : \quad a(u_H, v_H) = (f, v_H) \quad \forall v_H \in V_H,$$

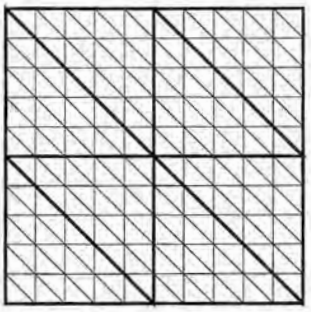

Figure 2.4 Coarse triangulation $\mathcal{T}^H$ and fine triangulation $\mathcal{T}^h$.

with a corresponding element stiffness matrix A_H.

Let n_H and n_h be the number of interior nodes of the coarse and fine triangulation, respectively. We linearly interpolate a coarse grid function $v_H \in V^H$ to a fine grid function $v_h \in V^h$ with the $n_h \times n_H$ matrix R_H^T. The transpose R_H of R_H^T is a weighted restriction map. Using these matrices, we can relate A_H to the element stiffness matrix A by

$$A_H = R_H A R_H^T.$$

Supplementary to the fine grid correction $\mathbf{z}^k = P_{as}^{-1}\mathbf{r}^k$ in steps 3 and 7 of Algorithm 2.1, we add a coarse grid correction

$$\mathbf{z}_H^k := R_H^T A_H^{-1} R_H \mathbf{r}^k.$$

This is similar to change the preconditioner P_{as} to

$$P_{as,H} = \left(R_H^T A_H^{-1} R_H + \sum_{i=1}^{M} B_i \right)^{-1}.$$

The use of the preconditoner $P_{as,H}$ results in a much better convergence behaviour of the PCG Algorithm 2.1. The condition number $\kappa(P_{as,H}^{-1}A)$ satisfies

$$\kappa(P_{as,H}^{-1}A) \leq C(1 + H/\delta), \tag{2.23}$$

where the constant C does not depend on H, h and δ. A proof of this result is given in (Dr94a).

3
IMPLEMENTATION AND RESULTS

In this chapter, we discuss the implementation of the additive Schwarz method on a distributed memory machine and present the results obtained by running various examples. The first section provides a brief introduction to the concepts of parallel computing, and will give an overview of the message-passing system MPI. The implementation using MPI is considered in detail in the second section. The numerical experiments are discussed in the third section.

3.1 Parallel computing with MPI

We implement the additive Schwarz method on a distributed memory machine. This hardware system may be classified as MIMD architecture, where MIMD stands for Multiple Instructions Multiple Data. Each processor has its own local memory and communicates with other processors through channels, as illustrated in Figure 3.1. A brief classification of different hardware systems can be found in (To96).

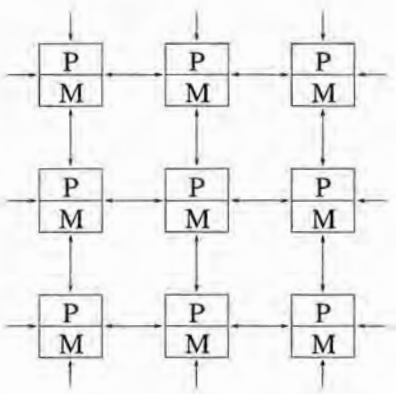

Figure 3.1 MIMD architecture with distributed memory.

We intend to partition the domain Ω into M subdomains Ω_i, $i = 1, \ldots, M$. In order to parallelize the M subdomain solution processes in steps 3 and 7 of Algorithm 2.1, the program runs on M processors. Processor p_i is responsible for the subdomain Ω_i, $i = 1, \ldots, M$.

We will see in the next section that we have to communicate between the processors for executing Algorithm 2.1 in parallel. The communications are performed using MPI, the message-passing interface. MPI provides the user with a library of routines which are called from the implemented program. During the last years, MPI has become a standard, and almost all vendors of distributed memory machines equip their systems with an MPI implementation. Thus, a code written using MPI is portable to many different platforms. A complete presentation of the MPI routines is given in (Sn98), an introductory book is (Gr94).

We give a brief overview of the MPI routines used in our implementation of the additive Schwarz method, and introduce some notation. For the sake of symplicity, we do not present the complete syntax of the routines.

- Let $\mathbf{v}^i$ denote a vector stored on processor p_i. The routine **MPI_send** is used to send the components of $\mathbf{v}^i$ from processor p_i to processor p_j. Processor p_j receives the components in the vector $\mathbf{v}^j$ using the routine **MPI_recv**. An MPI_send operation is only complete with its corresponding MPI_recv, and vice versa. We write $\texttt{send}(\mathbf{v}^i \longrightarrow p_j)$ for sending the components of $\mathbf{v}^i$ to processor p_j, and $\texttt{recv}(\mathbf{v}^j \longleftarrow p_i)$ for receiving the components in $\mathbf{v}^j$ from processor p_i.

- It will be necessary to reduce local vectors to global ones by taking either the sum or the componentwise maximum of the local vectors over all processors. This can be achieved with the routine **MPI_allreduce**. The global vector is placed on all processors. We write $\texttt{allreduce}(\mathbf{v}^i \longrightarrow \mathbf{v}, \texttt{sum})$ for reducing the vectors $\mathbf{v}^i$ to the vector $\mathbf{v}$ by taking the sum, and $\texttt{allreduce}(\mathbf{v}^i \longrightarrow \mathbf{v}, \texttt{max})$ for a reduction by taking the maximum. To illustrate this with a simple example, assume that the vector $\mathbf{v}^1 = (1, 3)^T$ is placed on processor p_1, and that $\mathbf{v}^2 = (4, 2)^T$ is placed on processor p_2. The operation $\texttt{allreduce}(\mathbf{v}^i \longrightarrow \mathbf{v}, \texttt{sum})$

places the vector $\mathbf{v} = \mathbf{v}^1 + \mathbf{v}^2 = (5,5)^T$ on both processors, and `allreduce`$(\mathbf{v}^i \longrightarrow \mathbf{v}, \texttt{max})$ reduces $\mathbf{v}^1$ and $\mathbf{v}^2$ to $\mathbf{v} = (\max\{1,4\}, \max\{3,2\})^T = (4,3)^T$.

3.2 Implementation of the additive Schwarz algorithm

As in the first chapter, we divide the algorithm into two basic steps.

1. The *initialization process*, consisting of
   (a) *decomposing* the domain Ω into M Subdomains,
   (b) *generating a mesh* of finite elements over each subdomain Ω_i, $i = 1, \ldots, M$, and
   (c) *assembly* of the local stiffness matrices A_{ii} and right hand sides $\mathbf{f}_i$, $i = 1, \ldots, M$.
2. *Solution* of the resulting system (1.15).

3.2.1 Domain decomposition

We want to partition the domain Ω into M subdomains such that the computational load per subdomain will be approximately the same. This can become a rather complicated task on an irregular geometry. There exists a variety of different approaches, see for instance (To96) and (Si91). We will only deal with rectangular domains which greatly simplifies the task.

Let Ω be a rectangle of width d_x and height d_y with lower left corner in the origin of the coordinate plane. Assume that the product Mh divides both d_x and d_y evenly, where h is the given mesh parameter. We perform a one-dimensional decomposition into M strips Ω_i', $i = 1, \ldots, M$, such that each strip Ω_i' has width $\frac{d_x}{M}$. The strips are numbered from left to right. The left and right sides of each strip are moved to form the overlapping subdomains Ω_i in accordance with the desired layers of overlap l. As we will see in the next Section 3.2.2, one layer of overlap has width h due to the uniform triangulation. Therefore, the overlapping regions must have a width of lh. We distinguish between the two cases when l is even and when l is odd. If l is even, the left side of Ω_i' is moved $\frac{lh}{2}$ to the left, $i = 2, \ldots, M$, and the right side is moved $\frac{lh}{2}$ to the right, $i = 1, \ldots, M-1$. If l is odd, the factors change to $\frac{l+1}{2}h$ and $\frac{l-1}{2}h$, respectively. Thus, in both cases the overlapping regions have width lh.

REMARK 3.1 The width of each subdomain Ω_i is a multiple of the mesh parameter h, since we assumed that Mh divides the width d_x of the domain Ω evenly, and because the width of the overlapping regions is a multiple of h. This fact will be necessary to guarantee the matching of the local triangulations in the overlapping regions.

Let us illustrate the process applied to the decomposition of the unit square into $M := 2$ subdomains (see Figure 3.2). Suppose that $h := \frac{1}{8}$ and $l := 2$. First, we get two strips of width $\frac{1}{2}$. The right side of Ω_1' is moved $\frac{lh}{2} = \frac{1}{8}$ to the right, and the left side of Ω_2' is moved $\frac{1}{8}$ to the left. Thus, we will get the desired 2 layers of overlap.

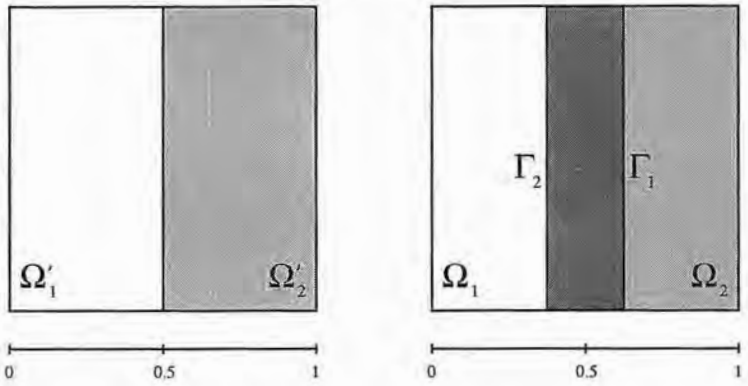

Figure 3.2 Decomposition of the unit square into 2 subdomains.

3.2.2 Mesh generation

On each subdomain Ω_i, the boundary nodes and interior nodes are generated in the same way as introduced in section 1.4.1. Following the algorithm, the boundary nodes are indexed starting in the lower left corner of the rectangle Ω_i going counterclockwise. Let n_i and $n_{b,i}$ indicate the number of interior nodes and boundary nodes, respectively, and set $m_i := n_{b,i} + n_i$. The coordinates of the nodes are stored in a $2 \times m_i$ matrix $\mathbf{P}$. The j-th column of $\mathbf{P}$ refers to the node j, the first and second row containing its x-coordinate and y-coordinate, respectively. A vector **localglobal** of length m_i which relates the local node numbers to the global ones is formed. Its j-th component is the number of node j in the global mesh.

The triangle generation portion of the algorithm introduced in section 1.4.1 turns out to be

very slow as the mesh parameter h becomes small. This is due to the fact that the algorithm selects always the nearest node among the available ones to form the apex of the new triangle. Therefore, the distance of the base segment to every available node is evaluated.

We make use of the rectangular geometry to gain performance, and set up the mesh information directly. Connecting the finite element nodes, we obtain a uniform triangulation $\mathcal{T}_i^h$ of $\overline{\Omega_i}$, see Figure 3.3. Due to the placement of the nodes, each triangle has width and height h. The number of triangles $n_{t,i}$ is calculated and a $3 \times n_{t,i}$ matrix $\mathbf{T}$ is formed. Each column of $\mathbf{T}$ contains the local node indices of the three vertices of one triangle.

Like the decomposition, the mesh generation is performed completely in parallel. There is no need for communication between the processors. Due to the fact mentioned in Remark 3.1, the local triangulations are guaranteed to match in the overlapping regions, and to induce a global triangulation $\mathcal{T}^h$ of the domain Ω.

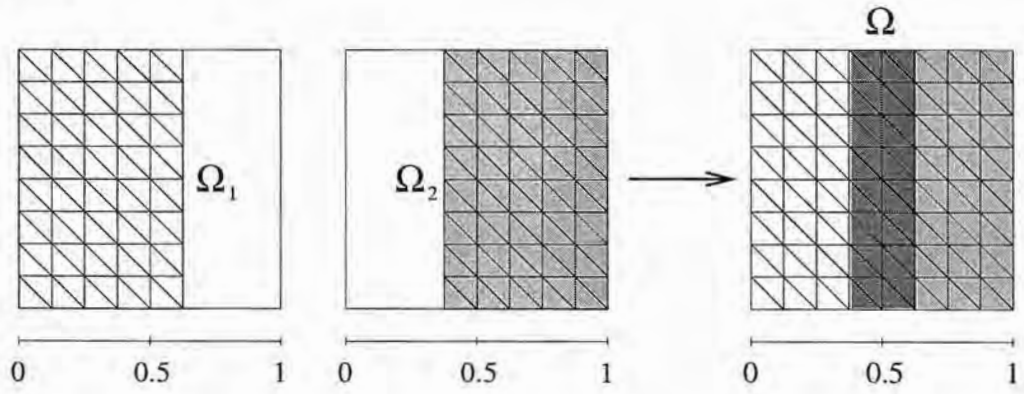

Figure 3.3 Uniform triangulation of the subdomains.

3.2.3 Assembly of the linear system

For the numerical experiments, we will only deal with the Poisson problem (1.1). Furthermore, we have the uniform triangulation $\mathcal{T}^h$ of the rectangle Ω. Summing up the contributions from each triangle $K \in \mathcal{T}^h$, the entries in the global element stiffness matrix A turn out to be

given by

$$a_{ij} = a(\varphi_i, \varphi_j) = \begin{cases} 4 & \text{for } j = i, \\ -1 & \text{for } j = i+1, j = i-1, j = i+n_v, j = i-n_v, \\ 0 & \text{else,} \end{cases}$$

where $n_v := \frac{d_y}{h} - 1$ indicates the number of interior nodes in the vertical direction.

Each processor p_i sets up its local element stiffness matrix A_{ii}. As mentioned in Section 2.4, the conjugate gradient method requires the calculation of the matrix-vector product $A\mathbf{p}$. In order to perform this operation in parallel, we need the submatrices $A_{i,i-1}$, if $i \in \{2, \ldots, M\}$, and $A_{i,i+1}$, if $i \in \{1, \ldots, M-1\}$, of the global stiffness matrix A. These $n_v \times n_v$ matrices are placed on processor p_i. The matrices represent the coupling of the nodes in the interior of Ω_i to the left and the right boundaries, respectively.

Figure 3.4 illustrates the resulting block structure of A applied to our example with $M = 2$. The matrices A_{12} and A_{21} represent the coupling of the nodes interior to the overlapping region $\Omega_1 \cap \Omega_2$ to the nodes on the boundary Γ_1 and Γ_2, respectively.

Practically, the processor p_i computes its corresponding n_i rows of the global stiffness matrix A. The processor p_i also calculates its corresponding n_i components of the right hand side $\mathbf{f}$ by summing up the contributions from the triangles $K \in \mathcal{T}_i^h$. As in section 1.4.2, we see that

$$\mathbf{f}_j = \sum_{K \in \mathcal{T}_i^h} \left(\int_K f \varphi_j \right), j = 1, \ldots, n_i.$$

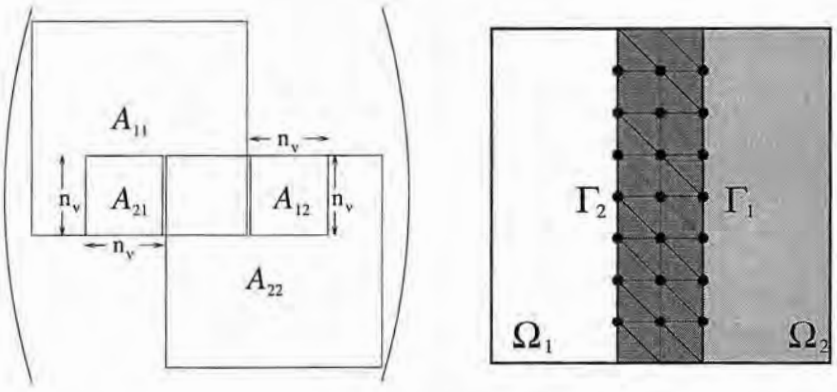

Figure 3.4 The block structure of A in the case of 2 subdomains.

The integral over the triangle K is computed using a well-known four-point formula. It is approximated with a weighted sum of the values of the integrand $f\varphi_j$ at 4 base points that lie within the triangle. A detailed description of the formula is given in (Po98).

REMARK 3.2 We saw earlier that the matrix A is sparse. In order to take advantage of the large number of zero elements, the sparse matrices are stored using a special scheme. By storing only the nonzero elements, much less memory space is needed and also the matrix-vector products can be computed faster. For our implementation, the so-called *compressed sparse row* (CSR) format is being used. An introduction to different storage schemes can be found in (Sa96).

3.2.4 Solution of the linear system

For simplicity, we restrict the discussion to the case of two subdomains. The generalization to the case of $M > 2$ subdomains is straightforward. We want to solve the linear system

$$A\mathbf{u} = \mathbf{f} \tag{3.1}$$

in parallel. The additive Schwarz method provides us with the preconditioner

$$P_{as} = (R_1^T A_{11}^{-1} R_1 + R_2^T A_{22}^{-1} R_2)^{-1}.$$

Let us recall Algorithm 2.1 used to solve the system (3.1).

ALGORITHM 3.1 (PRECONDITIONED CONJUGATE GRADIENT ALGORITHM)

1. Assign $\mathbf{u}^0$, and set
2. $\mathbf{r}^0 := \mathbf{f} - A\mathbf{u}^0$,
3. $\mathbf{p}^0 := \mathbf{z}^0 := P_{as}^{-1}\mathbf{r}^0$.

For $k \geq 0$ until convergence, calculate

4. $\alpha_k := \frac{(\mathbf{z}^k, \mathbf{r}^k)}{(\mathbf{p}^k, A\mathbf{p}^k)}$,
5. $\mathbf{u}^{k+1} := \mathbf{u}^k + \alpha_k \mathbf{p}^k$,
6. $\mathbf{r}^{k+1} := \mathbf{r}^k - \alpha_k A\mathbf{p}^k$,

7. $\mathbf{z}^{k+1} := P_{as}^{-1}\mathbf{r}^{k+1}$,
8. $\beta_{k+1} := \frac{(\mathbf{z}^{k+1},\mathbf{r}^{k+1})}{(\mathbf{z}^{k},\mathbf{r}^{k})}$,
9. $\mathbf{p}^{k+1} := \mathbf{z}^{k+1} + \beta_{k+1}\mathbf{p}^{k}$.

Let $\mathbf{f}^i := R_i\mathbf{f}$, $\mathbf{r}^{i,k} := R_i\mathbf{r}^k$, and $\mathbf{z}^{i,k} := R_i\mathbf{z}^k$ denote the restrictions of the vectors $\mathbf{f}$, $\mathbf{r}$, and $\mathbf{z}$ to the nodes in the interior of the subdomain Ω_i. Furthermore, denoting by n_h the number of nodes in the interior of Ω, let $\widehat{R_i}$ be the matrix whose action restricts a vector $\mathbf{v} \in \mathbf{R}^{n_h}$ to a vector of length $\overline{n_i} := n_i + n_v$ by preserving the components corresponding to the nodes in $\overline{\Omega_i} \cap \Omega$. Thus, the restricted vector includes the nodes on the artificial boundary Γ_i. Defining $\mathbf{u^{i,0}} := \widehat{R_i}\mathbf{u}^0$, and $\mathbf{p^{i,k}} := \widehat{R_i}\mathbf{r}^0$, we can compute the matrix-vector products $A\mathbf{u}^0$ and $A\mathbf{p}^k$ in parallel.

Let

$$A_1 := R_1 A \left(\widehat{R_1}\right)^T$$

be the $n_i \times \overline{n_i}$ matrix consisting of the local element stiffness matrix A_{11} in the upper left, and the matrix A_{12} in the lower right corner, and similarly, let

$$A_2 := R_2 A \left(\widehat{R_2}\right)^T,$$

see Figure 3.5. The matrix A_i is placed on processor p_i.

Corresponding to step 2 of Algorithm 3.1, we calculate the initial local residual, given by

$$\mathbf{r}^{i,0} := \mathbf{f}^i - A_i\mathbf{u}^{i,0},$$

on each processor.

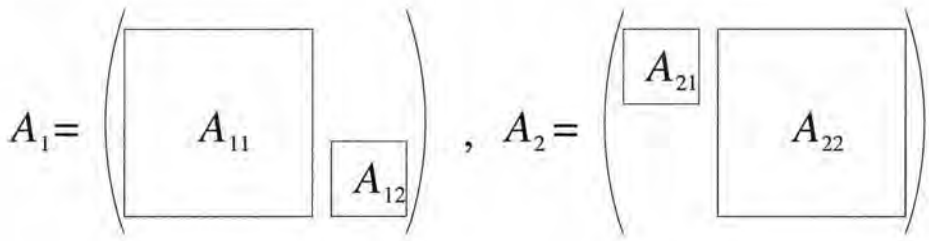

Figure 3.5 The matrices A_1 and A_2.

We now focus on calculating the correction $\mathbf{z}^k$ in steps 3 and 7. Let n_c indicate the number of common nodes, namely, the nodes in the interior of both Ω_1 and Ω_2, and set $n_i^0 := n_i - n_c$, i.e., the number of nodes belonging exclusively to the interior of Ω_i. Note that the last n_c components of the vector $\mathbf{z}^{1,k}$ and the first n_c components of $\mathbf{z}^{2,k}$ correspond to the common nodes. First, each processor p_i computes

$$\widetilde{\mathbf{z}}^{i,k} := A_{ii}^{-1}\mathbf{r}^{i,k}.$$

We obtain $\widetilde{\mathbf{z}}^{i,k}$ by solving the system

$$A_{ii}\widetilde{\mathbf{z}}^{i,k} = \mathbf{r}^{i,k}$$

with an exact or inexact solver. Then, processor p_1 sends $\widetilde{\mathbf{z}}^{1,k}_{(n_1^0+1,\ldots,n_1)}$ to processor p_2, and processor p_2 sends $\widetilde{\mathbf{z}}^{2,k}_{(1,\ldots,n_c)}$ to processor p_1. The local correction is then computed on p_1 by

$$\mathbf{z}_j^{1,k} := \begin{cases} \widetilde{\mathbf{z}}_j^{1,k} & \text{for } j = 1,\ldots,n_1^0 \\ \widetilde{\mathbf{z}}_j^{1,k} + \widetilde{\mathbf{z}}_{j-n_1^0}^{2,k} & \text{for } j = n_1^0+1,\ldots,n_1, \end{cases}$$

and similarly on p_2 by

$$\mathbf{z}_j^{2,k} := \begin{cases} \widetilde{\mathbf{z}}_j^{2,k} & \text{for } j = n_c+1,\ldots,n_2 \\ \widetilde{\mathbf{z}}_j^{2,k} + \widetilde{\mathbf{z}}_{j+n_1^0}^{1,k} & \text{for } j = 1,\ldots,n_c, \end{cases}$$

When the vector $\mathbf{p}^k$ has to be computed in steps 3 and 9, processor p_1 can calculate only the first n_1 components of $\mathbf{p}^{1,k}$ correctly, and p_2 only the last n_2 components of $\mathbf{p}^{2,k}$. Note that

$$\mathbf{p}^{1,k}_{(n_1+1,\ldots,n_1+n_v)} = \mathbf{p}^{2,k}_{(n_c+n_v+1,\ldots,n_c+2n_v)},$$

and

$$\mathbf{p}^{2,k}_{(1,\ldots,n_v)} = \mathbf{p}^{1,k}_{(n_1^0-n_v+1,\ldots,n_1^0)}.$$

Therefore, the processors exchange the corresponding components using the MPI routines `send` and `recv`.

The constants α_k and β_k are needed on both processors. Each processor computes the local components of the dotproducts involved in steps 4 and 8. Then, the global dotproducts are

placed on each processor with the MPI routine `allreduce`. To avoid that the products of the components corresponding to the common nodes are added twice, processor p_1 takes only the first n_1^0 components of the involved vectors for the calculation. For instance,

$$(\mathbf{z}^k, \mathbf{r}^k) = (\mathbf{z}^{1,k}_{(1,\ldots,n_0^1)}, \mathbf{r}^{1,k}_{(1,\ldots,n_0^1)}) + (\mathbf{z}^{2,k}, \mathbf{r}^{2,k}).$$

In order to check for convergence, we follow the practice in (Sm96). Convergence is declared when the infinity norm of the initial residual is decreased by three orders in magnitude, namely,

$$\frac{\left\|\mathbf{r}^{k+1}\right\|}{\|\mathbf{r}^0\|} < \frac{1}{1000}.$$

To place the infinity norm of the global residual $\mathbf{r}^k$ on both processors, the MPI routine `allreduce` is used after calculating the local residuals $\mathbf{r}^{i,k}$.

We are now able to present the parallel implementation of the preconditioned conjugate gradient algorithm in compact form.

ALGORITHM 3.2 (PARALLEL PRECONDITIONED CONJUGATE GRADIENT ALGORITHM)

1. Assign $\mathbf{u}^{i,0}$ on both processors.
2. $\mathbf{r}^{i,0} := \mathbf{f}^i - A_i\mathbf{u}^{i,0}$

   `allreduce` $(\|\mathbf{r}^{i,0}\| \longrightarrow \|\mathbf{r}^0\|,$ `max`$)$
3. p_1: $\tilde{\mathbf{z}}^{1,0} := A_{11}^{-1}\mathbf{r}^{1,0}$

   `send` $(\tilde{\mathbf{z}}^{1,0}_{(n_1^0-n_v+1,\ldots,n_1)} \longrightarrow p_2)$

   `recv` $(\mathbf{p}^{1,0}_{(n_1+1,\ldots,\overline{n_1})},\ \tilde{\mathbf{z}}^{2,0}_{(1,\ldots,n_c)} \longleftarrow p_2)$

   $\mathbf{z}^{1,0}_{(1,\ldots,n_1^0)} := \tilde{\mathbf{z}}^{1,0}_{1,\ldots,n_1^0}$

   $\mathbf{z}^{1,0}_{(n_1^0+1,\ldots,n_1)} := \tilde{\mathbf{z}}^{1,0}_{(n_1^0+1,\ldots,n_1)} + \tilde{\mathbf{z}}^{2,0}_{(1,\ldots,n_c)}$

   $\mathbf{p}^{1,0}_{(1,\ldots,n_1)} := \mathbf{z}^{1,0}$

   p_2: $\tilde{\mathbf{z}}^{2,0} := A_{22}^{-1}\mathbf{r}^{2,0}$

   `send` $(\tilde{\mathbf{z}}^{2,0}_{(1,\ldots,n_c+n_v)} \longrightarrow p_1)$

   `recv` $(\mathbf{p}^{2,0}_{(1,\ldots,n_v)},\ \tilde{\mathbf{z}}^{1,0}_{(n_1^0+1,\ldots,n_1)} \longleftarrow p_1)$

   $\mathbf{z}^{2,0}_{(n_c+1,\ldots,n_2)} := \tilde{\mathbf{z}}^{2,0}_{(n_c+1,\ldots,n_2)}$

   $\mathbf{z}^{2,0}_{(1,\ldots,n_c)} := \tilde{\mathbf{z}}^{2,0}_{(1,\ldots,n_c)} + \tilde{\mathbf{z}}^{1,0}_{(n_1^0+1,\ldots,n_1)}$

   $\mathbf{p}^{2,0}_{(n_v+1,\ldots,\overline{n_2})} := \mathbf{z}^{2,0}$

For $k \geq 0$ until convergence solve

4. p_1: $\mathbf{zr}^{1,k} := \left(\mathbf{z}^{1,k}_{(1,\ldots,n_1^0)}, \mathbf{r}^{1,k}_{(1,\ldots,n_1^0)}\right)$

$\mathbf{pAp}^{1,k} := \left(\mathbf{p}^{1,k}_{(1,\ldots,n_1^0)}, (A_1\mathbf{p}^{1,k})_{(1,\ldots,n_1^0)}\right)$

p_2: $\mathbf{zr}^{2,k} := \left(\mathbf{z}^{2,k}, \mathbf{r}^{2,k}\right)$

$\mathbf{pAp}^{2,k} := \left(\mathbf{p}^{2,k}_{(n_v+1,\ldots,n_2)}, A_2\mathbf{p}^{2,k}\right)$

allreduce $(\mathbf{zr}^{i,k} \longrightarrow \mathbf{zr}^k,$ sum)

allreduce $(\mathbf{pAp}^{i,k} \longrightarrow \mathbf{pAp}^k,$ sum)

$\alpha_k := \frac{\mathbf{zr}^k}{\mathbf{pAp}^k}$

5. $\mathbf{u}^{i,k+1} := \mathbf{u}^{i,k} + \alpha_k \mathbf{p}^{i,k}$

6. $\mathbf{r}^{i,k+1} := \mathbf{r}^{i,k} - \alpha_k A\mathbf{p}^{i,k}$

allreduce $(\|\mathbf{r}^{i,k+1}\| \longrightarrow \|\mathbf{r}^{k+1}\|,$ max)

If $\frac{\|\mathbf{r}^{k+1}\|}{\|\mathbf{r}^0\|} < \frac{1}{1000}$, stop.

7. p_1: $\widetilde{\mathbf{z}}^{1,k+1} := A_{11}^{-1}\mathbf{r}^{1,k+1}$

send $(\widetilde{\mathbf{z}}^{1,k+1}_{(n_1^0+1,\ldots,n_1)} \longrightarrow p_2)$

recv $(\widetilde{\mathbf{z}}^{2,k+1}_{(1,\ldots,n_c)} \longleftarrow p_2)$

$\mathbf{z}^{1,k+1}_{(1,\ldots,n_1^0)} := \widetilde{\mathbf{z}}^{1,k+1}_{1,\ldots,n_1^0}$

$\mathbf{z}^{1,k+1}_{(n_1^0+1,\ldots,n_1)} := \widetilde{\mathbf{z}}^{1,k+1}_{(n_1^0+1,\ldots,n_1)} + \widetilde{\mathbf{z}}^{2,k+1}_{(1,\ldots,n_c)}$

p_2: $\widetilde{\mathbf{z}}^{2,k+1} := A_{22}^{-1}\mathbf{r}^{2,k+1}$

send $(\widetilde{\mathbf{z}}^{2,k+1}_{(1,\ldots,n_c)} \longrightarrow p_1)$

recv $(\widetilde{\mathbf{z}}^{1,k+1}_{(n_1^0+1,\ldots,n_1)} \longleftarrow p_1)$

$\mathbf{z}^{2,k+1}_{(n_c+1,\ldots,n_2)} := \widetilde{\mathbf{z}}^{2,k+1}_{(n_c+1,\ldots,n_2)}$

$\mathbf{z}^{2,k+1}_{(1,\ldots,n_c)} := \widetilde{\mathbf{z}}^{2,k+1}_{(1,\ldots,n_c)} + \widetilde{\mathbf{z}}^{1,k+1}_{(n_1^0+1,\ldots,n_1)}$

8. p_1: $\mathbf{zr}^{1,k+1} := \left(\mathbf{z}^{1,k+1}_{(1,\ldots,n_1^0)}, \mathbf{r}^{1,k+1}_{(1,\ldots,n_1^0)}\right)$

p_2 : $\mathbf{zr}^{2,k+1} := \left(\mathbf{z}^{2,k+1}, \mathbf{r}^{2,k+1}\right)$

allreduce $(\mathbf{zr}^{i,k+1} \longrightarrow \mathbf{zr}^{k+1},$ sum)

$\beta_{k+1} := \frac{\mathbf{zr}^{k+1}}{\mathbf{zr}^k}$

Increasing the size of the overlapping regions results in a better convergence behaviour. We see this in Table 3.3 below which shows the number of iterations as we vary the overlap l for different numbers of subdomains M. No coarse grid correction is applied. The mesh parameter h is equal to $\frac{1}{64}$. Note that the largest improvement is made changing from one layer of overlap to two layers. This is due to the fact that for one layer the subdomains do not share any internal nodes.

Table 3.3 Variation of the overlap.

	M			
overlap	**2**	4	**8**	**16**
1	21	22	27	36
2	15	15	19	26
3	11	13	15	21
4	10	11	13	17

We are able to verify another implication of the fact that the condition number $\kappa(P_{as}^{-1}A)$ is on the order of $1/H\delta$. When we keep δ and the number of subdomains M constant, the number of required iterations turns out to be very insensitive to changes of h, i.e., to refinements of the triangulation $\mathcal{T}^h$. We start with $h = \frac{1}{16}$ and one layer of overlap, i.e., $\delta = h$, and for each following execution of the algorithm we divide h by 2 and double l. Thus, $\delta = \frac{1}{16}$ is kept constant. The results for different numbers of subdomains are documented in Table 3.4.

Table 3.4 Iteration count with constant δ.

		M			
h	overlap	**2**	4	**8**	**16**
$\frac{1}{16}$	1	10	11	15	21
$\frac{1}{32}$	2	11	11	14	19
$\frac{1}{64}$	4	10	11	13	17
$\frac{1}{128}$	8	9	10	13	16

9. p_1: $\mathbf{p}^{1,k+1}_{(1,\ldots,n_1)} := \mathbf{z}^{1,k+1} + \beta_{k+1}\mathbf{p}^{1,k}_{(1,\ldots,n_1)}$
   send $(\mathbf{p}^{1,k+1}_{(n_1^0-n_v+1,\ldots,n_1^0)} \longrightarrow p_2)$
   recv $(\mathbf{p}^{1,k+1}_{(n_1+1,\ldots,\overline{n_1})} \longleftarrow p_2)$
   p_2: $\mathbf{p}^{2,k+1}_{(n_v+1,\ldots,\overline{n_2})} := \mathbf{z}^{2,k+1} + \beta_{k+1}\mathbf{p}^{2,k}_{(n_v+1,\ldots,\overline{n_2})}$
   send $(\mathbf{p}^{2,k+1}_{(n_c+1,\ldots,n_c+n_v)} \longrightarrow p_1)$
   recv $(\mathbf{p}^{2,k+1}_{(1,\ldots,n_v)} \longleftarrow p_1)$

3.2.5 Coarse grid correction

If we intend to apply the two-level additive Schwarz method (see Section 2.5), we need to get the triangulation $\mathcal{T}^H$, and to assemble the stiffness matrix A_H and the right hand side $\mathbf{f}_H$ of the coarse system

$$A_H \mathbf{u}_H = \mathbf{f}_H. \tag{3.2}$$

We set the mesh parameter H for the coarse grid equal to d_x/M, the width of the domain Ω divided by the number of subdomains. The generation of the coarse mesh and the assembly of the linear system (3.2) is performed using the same scheme as mentioned in the sections above. Due to the asumption that hM divides both d_x and d_y, the mesh parameter H is a multiple of the mesh parameter h for the fine grid. Hence, the triangulation $\mathcal{T}^h$ is a refinement of $\mathcal{T}^H$.

The interpolation matrix R_H^T is obtained using a simple linear interpolation formula, see for instance (Po98). Each processor p_i calculates the n_i rows of R_H^T which correspond to the interior nodes of the subdomain Ω_i, and obtains

$$R_{H,i}^T := R_i R_H^T.$$

In steps 3 and 7 of Algorithm 3.2, the residual $\mathbf{r}_H^k$ of the coarse system is calculated by

$$\mathbf{r}_H^k := R_H \mathbf{r}^k.$$

The computation above is performed in parallel, by adding up the contributions from the local matrix-vector products $R_{H,i}\mathbf{r}^{i,k}$, using the MPI routine `allreduce`. Then, the coarse system is solved, and the solution is interpolated on each processor, deriving

$$\mathbf{z}_H^{i,k} := R_{H,i}^T A_H^{-1} \mathbf{r}_H^k.$$

The local coarse grid corrections $\mathbf{z}_H^{i,k}$ are added to the fine grid corrections $\mathbf{z}^{i,k}$.

3.3 Numerical experiments

For the following experiments, the additive Schwarz algorithm is applied to the model problem

$$-\Delta u = 1 \quad \text{in } \Omega, \tag{3.3}$$

$$u = 0 \quad \text{on } \Gamma, \tag{3.4}$$

where Ω is the square of side length 2.

3.3.1 Variation of the overlap and of the number of subdomains

We investigate the number of iterations needed until convergence can be declared (see step 6 of Algorithm 3.2). The subdomain problems are solved with an exact solver, namely, by using the conjugate gradient algortihm.

As mentioned earlier, the condition number $\kappa(P_{as}^{-1}A)$ is on the order of $1/H\delta$, where H is the diameter of the largest subdomain and δ the width of the smallest overlapping region. This suggests that we have to expect slower convergence, as the number of subdomains M becomes larger, and H becomes smaller. Table 3.1 shows the number of iterations until convergence for different values of h and M. The number of layers of overlap l is 2. Those entries are left blank where the triangulation is too coarse for a decomposition into the specified number of subdomains. The last row contains the iteration count for solving the linear system (3.1) with the unpreconditioned CG Algorithm 1.1. The columns of Table 3.1 show that indeed the required number of iterations increases as M becomes larger. Note the significant decrease in

Table 3.1 Iteration count for the additive Schwarz method.

	h			
M	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$
2	7	11	15	20
4	9	11	15	22
8	11	14	19	25
16	15	19	26	35
32	—	27	35	46
64	—	—	47	65
CG	39	72	134	257

the number of iterations compared to the standard CG method. Looking at each line separately, we also see that for a fixed number of subdomains the number of iterations increases as the mesh parameter h becomes smaller. This is due to the fact that the width of the overlapping regions $\delta = lh$ becomes smaller.

We repeat the same experiment, but this time the coarse grid correction is applied, as described in Section 3.2.5. This results in a significant decrease in iterations, as documented in Table 3.2. Especially in the last column, we see that the number of iterations becomes smaller as the number of subdomains M increases. This is due to the fact that the coarse triangulation $\mathcal{T}^H$ gets finer and hence, the coarse grid correction becomes more accurate. We see this effect in the estimate for the condition number (2.23), where the term H/δ decreases. Note also that the convergence behaviour is less affected by refinements of the triangulation, when the number of subdomains M is kept constant.

Table 3.2 Iterations for the two-level method.

	h			
M	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$
2	7	8	9	17
4	6	7	9	13
8	6	6	7	10
16	—	6	6	8
32	—	—	6	6
64	—	—	—	6
CG	39	72	134	257

3.3.2 Speedup

We analyze the speedup which is obtained executing the program with different numbers of processors. Various different performance evaluation models can be found in today's literature, see for example (Wu98). We intuitively define speedup as the ratio of serial execution time to parallel execution time.

Let $T_{1,M}$ denote the serial execution time, namely, the time needed by a single processor to execute the initialization process and Algorithm 3.1, taking the preconditioner P_{as} based on M subdomains. For calculating the corrections $\mathbf{z}^k$ in steps 3 and 7 of Algorithm 3.1, the single processor works on each subdomain in a sequential process. We indicate by T_M the parallel execution time for running the program with M processors, that is, we take the maximum over the times needed by each of the processors. Table 3.5 presents the results $T_{1,M}/T_M$, using different mesh parameters h.

An ideal parallelization would result in a linear speedup, namely, $T_{1,M}/T_M = M$. In practice, linear speedup cannot be achieved in most applications, since the processors need to exchange information. The ratio of communication time to computation time increases with the number of processors. We can see this effect very clearly in Table 3.5 for the case of $h = \frac{1}{64}$. For 2, 4, and 8 processors the speedup is almost ideal. As the number of processors M increases further, the communication time becomes a more important factor and affects the speedup negatively. Let us visualize the speedup in Figure 3.6. The dashed line indicates the ideal speedup, the solid line the result from the experiment.

Table 3.5 Speedup of the parallel execution compared to the serial execution.

	h			
M	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$
2	1.9	2.1	2.0	1.8
4	3.4	4.1	3.9	3.9
8	6.1	7.0	7.8	7.6
16	12.6	12.0	14.0	14.8
32	—	24.8	24.7	30.0
64	—	—	56.5	54.8

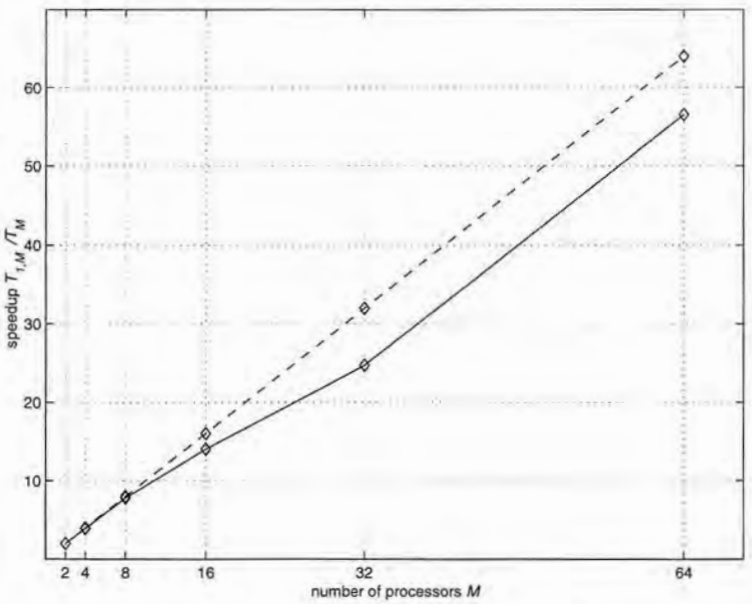

Figure 3.6 Ideal and realized speedup.

3.3.3 PCG versus CG

Let us finally compare the time needed for the execution of Algorithm 3.2, denoted by $\widehat{T}_M$, with the time needed by a parallelized version of the unpreconditioned conjugate gradient method (Algorithm 1.1), indicated by T_M^u. Algorithm 1.1 is parallelized by distributing the required calculations of the involved matrix-vector products and dotproducts over M processors, similarly as in Algorithm 3.2. Note that for a fixed mesh parameter h the number of iterations remains constant when the number of processors M is being changed. The initialization process is not timed.

If exact solvers are used on the subdomains, it may happen that $\widehat{T}_M > T_M^u$. As mentioned in Remark 2.2, it is more efficient to use inexact solvers. We approximate the local stiffness matrices A_{ii} by the well-known SSOR preconditioner. A detailed discussion of the SSOR method, i.e., Symmetric Successive OverRelaxation method, can be found in (Sa96).

If D indicates the diagonal of A_{ii}, $-E$ its strict lower part, and $-F$ its strict upper part, we approximate A_{ii} by

$$\mathcal{A}_{ii} := (D - \omega E)D^{-1}(D - \omega F).$$

The parameter ω is chosen as 1.8, which turns out to result in a good convergence behaviour. The computation of the local corrections $\mathbf{z}^{i,k} = \mathcal{A}_{ii}^{-1}\mathbf{r}^{i,k}$ in steps 3 and 7 of Algorithm 3.2 involves only the simple and fast solution of an upper and a lower triangular system.

The speedup $\widehat{T}_M/T_M^u$ is documented in Table 3.6. No coarse grid correction is used, the number of layers of overlap l is equal to 2. The Schwarz method is up to four times faster than the parallel unpreconditioned CG algorithm.

Table 3.6 Speedup of the Schwarz method compared to the parallel CG algorithm.

	h			
M	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$
2	2.3	3.1	3.8	4.3
4	1.7	2.7	3.7	4.3
8	1.7	2.5	3.4	3.9
16	1.4	1.9	2.7	3.4
32	—	1.7	2.2	2.9
64	—	—	1.7	2.4

We also make two other observations:

- Each row shows that the speedup increases as the triangulation $\mathcal{T}^h$ is refined. This is due to the fact that the CG algorithm is more sensitive towards refinements, and the iteration count is growing faster than the one for the preconditioned method.

- The speedup becomes less as the number of subdomains is increased. This is a consequence of the growing number of iterations for the Schwarz method, whereas the iteration count for the CG method remains constant.

We use the coarse grid correction in order to strengthen the first effect and to weaken the second one. The coarse system is obtained as described in Section 3.2.5. To gain performance, we want to avoid that the coarse triangulation becomes too fine, and the coarse system too expensive to solve. Therefore, the mesh parameter H is set equal to $d_x/8$, if the number of processors becomes larger than 8. Table 3.7 presents the results. As the problem size becomes large, the coarse grid correction results in a better speedup than the one achieved by the one-

level method. For the case of $h = \frac{1}{128}$, documented in the last column, the two-level Schwarz method is up to seven times faster than the parallel CG algorithm.

Table 3.7 Speedup of the two-level Schwarz method compared to the parallel CG algorithm.

	h			
M	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$
2	2.6	4.1	5.0	6.0
4	1.8	3.5	4.8	7.4
8	0.9	2.4	4.6	7.0
16	—	2.1	4.0	6.8
32	—	—	3.5	6.2
64	—	—	—	4.5

Let us visualize the achieved speedup of the Schwarz method compared to the CG algorithm in Figure 3.7. As example, the mesh parameter $h = \frac{1}{128}$ was chosen. The dashed line shows the performance gain for the additive Schwarz method without coarse grid correction, the solid line indicates the speedup for the two-level method.

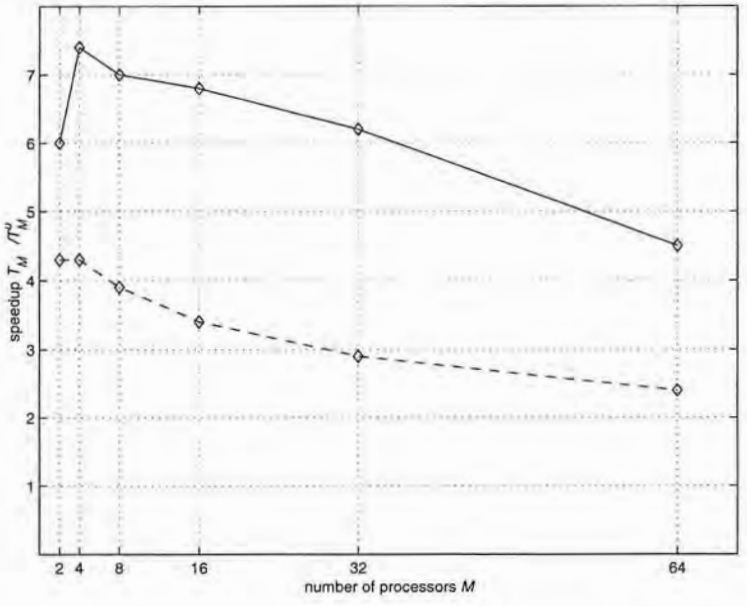

Figure 3.7 Comparison to the unpreconditioned CG algorithm.

APPENDIX A
FUNCTION SPACES

The definitions of some function spaces which are frequently used throughout the thesis are reviewed. Complete presentations of this subject can be found for instance in (Ad75) and (Ku77).

Hilbert and Banach spaces

Let V be a real linear space. An *inner product* on V is a map $(\cdot,\cdot) : V \times V \to \mathbf{R}$ which is bilinear, symmetric and *positive definite*, namely, $(v,v) \geq 0$ for all $v \in V$, and $(v,v) = 0$ implies $v = 0$. A *seminorm* is a map $||\cdot|| : V \to \mathbf{R}$ such that $||v|| \geq 0$ for all $v \in V$, $||cv|| = |c| \cdot ||v||$ for all $c \in \mathbf{R}$ and $v \in V$, and $||v+w|| \leq ||v|| + ||w||$ for all $v, w \in V$ (triangle inequality). A *norm* on V is a seminorm satisfying the additional property that $||v|| = 0$ implies $v = 0$. Any inner product defines a norm by setting $||v|| := (v,v)^{1/2}$.

A linear space V endowed with an inner product (respectively, a norm) is called a *pre-hilbertian* (respectively, *normed*) space. A sequence v_n is a Cauchy sequence in a normed space V provided that it is a Cauchy sequence with respect to the distance $d(v,w) := ||v-w||$. If any Cauchy sequence in a pre-hilbertian (respectively, normed) space V converges to an element of V, the space V is called a *Hilbert* space (respectively, *Banach* space).

L^p spaces

Let Ω be an open set contained in $\mathbf{R}^d$, $d \geq 1$, endowed with the Lebesgue measure. Let $1 \leq p < \infty$ and consider the set of measurable functions v such that

$$\int_\Omega |v|^p < \infty.$$

This space is denoted by $L^p(\Omega)$. In $L^p(\Omega)$ two functions which are different on a subset of measure zero are identified with each other. $L^p(\Omega)$ is a Banach space with respect to the norm

$$||v||_{L^p(\Omega)} := \left(\int_\Omega |v|^p\right)^{1/p}.$$

The space $L^2(\Omega)$ is a Hilbert space, endowed with the inner product

$$(v, w) := \int_\Omega vw.$$

Sobolev spaces

Let $v \in L^p(\Omega)$ and $\alpha = (\alpha_1, \ldots, \alpha_d)$ be a *multi-index* with each α_i a non-negative integer. We use the notation

$$D^\alpha v := \frac{\partial^{|\alpha|} v}{\partial x_1^{\alpha_1} \ldots x_d^{\alpha_d}},$$

where $|\alpha| := \alpha_1 + \ldots + \alpha_d$ is the *length* of α. The *Sobolev* space $W^{k,p}(\Omega)$, k a non-negative integer and $1 \leq p < \infty$, is the space of functions $v \in L^p(\Omega)$ such that all the derivatives $D^\alpha v$ of order up to k belong to $L^p(\Omega)$,

$$W^{k,p}(\Omega) := \{v \in L^p(\Omega) : D^\alpha v \in L^p(\Omega) \text{ for each non-negative multi-index } \alpha \text{ such that } |\alpha| \leq k\}.$$

$W^{k,p}(\Omega)$ is a Banach space with respect to the norm

$$||v||_{k,p,\Omega} := \left(\sum_{|\alpha| \leq k} ||D^\alpha v||^p_{L^p(\Omega)}\right)^{1/p}.$$

Moreover, a seminorm is defined by

$$|v|_{k,p,\Omega} := \left(\sum_{|\alpha| = k} ||D^\alpha v||^p_{L^p(\Omega)}\right)^{1/p}.$$

In particular, when $p = 2$ we write $H^k(\Omega)$ instead of $W^{k,2}(\Omega)$, $||\cdot||_{k,\Omega}$ and $|\cdot|_{k,\Omega}$ instead of $||\cdot||_{k,2,\Omega}$ and $|\cdot|_{k,2,\Omega}$, respectively.

Let $C_0^\infty(\Omega)$ denote the space of infinitely often differentiable functions having *compact support*, i.e., vanishing outside a bounded open set $\Omega' \subset \Omega$ which has a positive distance from the boundary Γ of Ω. Then $W_0^{k,p}(\Omega)$ denotes the closure of $C_0^\infty(\Omega)$ with respect to the norm $||\cdot||_{k,2,\Omega}$. When $p = 2$, we write $H_0^k(\Omega)$ instead of $W_0^{k,2}(\Omega)$.

APPENDIX B
FORTAN CODE

The fortran code is divided into three files. The file "pas.f90" contains the actual program PARALLEL_ADDITIVE_SCHWARZ, including the input routine, the assembly routines, the parallel CG and PCG algorithm, and also the PCG algorithm with the additive Schwarz preconditioner used for serial execution. The file "meshmod.f90" contains the module MESH_MODULE and consists of the subroutines used to perform the triangulation of the domain. The module SPARSE_MODULE, included in "sparse.f90", contains the sparse matrix routines and the serial conjugate gradient algorithms for the subdomain solves. The three files are listed in the next sections.

The program starts with the input routine. It asks for the mesh parameter h, which should be selected in form of $1/2^k$, k a positive integer. If one processor is used, a choice can be made whether to use the standard CG algorithm or the preconditioned one with a specified number of subdomains M. In this case the PCG algorithm selects 2 layers of overlap, solves exactly on the subdomains, and does not apply a coarse grid correction. If two or more processors are used (preferrably in form of 2^n), it can be selected whether to use the standard or the precondtioned CG algorithm. For the Schwarz algorithm, one may choose whether to apply a coarse grid correction or not. In the latter case, the number of layers of overlap may be specified. If a coarse grid correction is used, the algorithm selects 2 layers. Finally, one can specify the type of solvers used on the subdomains (exact or inexact).

The program produces a file "output.txt" which consists of the matrices $\mathbf{P}$, $\mathbf{E}$, and $\mathbf{T}$, and the solution vector $\mathbf{u}$. The matrices contain the mesh information. The MATLAB file "matmesh.m" can be used to read "output.txt" and to visualize the mesh and the solution. If

MATLAB is used, the mesh parameter should not be less than 1/8. Otherwise, the size of the matrices becomes too big.

B.1 pas.f90

```
    program PARALLEL_ADDITIVE_SCHWARZ
!*****************************************************************************
! Variables:
!    cache, ncache, nflush, and flush are used to flush the cache in order to
!    get correct timing results.
!    p: the number of processors used
!    rank: the rank of the processor executing the program, 0,...,p-1
!    status, ierror: for the MPI routines
!    f: the right hand side of the Poisson equation
!    u0: the function specifying the Dirichlet boundary condition
!    Lu0: Laplace operator applied to u0
! global variables:
!    diamx, diamy: width and height of the rectangular domain
!    h: the mesh parameter for the triangulation
!    prec: = 0 --> standard CG algorithm
!          = 1 --> preconditioned CG algorithm
!    coarse: = 0 --> no coarse grid correction
!            = 1 --> coarse grid correction
!    overlap: layers of overlap
!    exact: = 0 --> inexact subdomain solves
!           = 1 --> exact subdomain solves
!    M: number of subdomains, if the PCG algorithm is executed in serial
!    omega: relaxation parameter for the SSOR precond. for the local solves
!    gl_nbnodes: the number of boundary nodes in the global mesh
!    it: iteration counter
!    ntrials: number of runs
!    tmax: the maximum over the times needed by each processor for one run
!    tavg: the average time for ntrials runs
! local variables:
!    nodes: number of nodes including boundary nodes
!    intnodes: number of interior nodes
!    elements: number of elements
!    coord: matrix which contains the coordinates of the nodes
!    elem: matrix which contains the triangle information
!    localglobal: vector which relates the local node indices to the global ones
!    Stiff: the finite element stiffness matrix
!    rhs: the load vector
!    u, uint: solution vectors
!    t: time needed for one run
!*****************************************************************************

    use MESH_MODULE
    use SPARSE_MODULE
    integer, parameter:: cache = 96, ncache = cache*1024/8, nflush = ncache*4
    integer:: p, rank, status(MPI_STATUS_SIZE), ierror
    integer:: prec, coarse, overlap, exact, M
    integer:: nodes, intnodes, gl_nbnodes, elements, k, it, ntrials
    integer, pointer:: localglobal(:), elem(:,:)
    real*8:: diamx, diamy, h, omega, t, tmax, tavg, flush(nflush)
    real*8, pointer:: coord(:,:)
    real*8, allocatable:: u(:), rhs(:), x0vec(:), uint(:)
```

```
type(sparsematrix):: Stiff

f(x,y) = 1.0
u0(x,y) = 0.0
Lu0(x,y) = 0.0
diamx = 2.0
diamy = 2.0
omega = 1.8
ntrials = 5

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, p, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)


call INPUT(h, prec, coarse, overlap, exact, M, p, rank)


tavg = 0.0
do k = 1, ntrials
   if (p .gt. 1) then
      call DECOMPOSE_AND_MESH(diamx, diamy, h, overlap, coord, nodes, &
                              localglobal, elem, elements, gl_nbnodes, &
                              intnodes, xmin, xmax, p, rank)
   else
      call MESH_RECTANGLE(0.0, diamx, diamy, h, coord, nodes, elem, &
                          elements)
      intnodes = NINT((diamx/h - 1)*(diamy*h - 1))
      gl_nbnodes = nodes - intnodes
      allocate(localglobal(nodes))
      do i = 1, nodes
         localglobal(i) = i
      enddo
   endif


   allocate(u(nodes), rhs(intnodes))
   call ASSEMBLE(coord, nodes, intnodes, gl_nbnodes,elem, elements,&
                 h, localglobal, xmin, diamy, rank, p, rhs, Stiff)


   call random_number(flush)
   flush = flush + 0.0123
   call mpi_barrier(MPI_COMM_WORLD, ierror)
   t = mpi_wtime()
   if (p .eq. 1) then
      if (prec .eq. 1) then
         call SOLVE_SERIAL(Stiff, intnodes, rhs, nodes, overlap, &
                           u, h, it, diamx, diamy, omega, M)
      else
         allocate(x0vec(intnodes), uint(intnodes))
         x0vec = 0.1
         call CG(Stiff, rhs, x0vec, uint, 1000)
         do i = 1, nodes
            u(i) = u0(coord(i,1),coord(i,2))
         enddo
         u(nodes-intnodes+1:nodes) = u(nodes-intnodes+1:nodes) + uint
```

```
      endif
   else if (prec .eq. 1) then
      call PARALLEL_PCG(Stiff, intnodes, rhs, coord, nodes, overlap, &
                        exact, coarse, omega, gl_nbnodes, u, &
                        h, it, diamx, diamy, rank, p)
   else
      call PARALLEL_CG(Stiff, intnodes, rhs, nodes, overlap, &
                       gl_nbnodes, u, h, it, diamx, diamy, rank, p)
   endif
   t = mpi_wtime() - t
   call mpi_barrier(MPI_COMM_WORLD, ierror)


   if (p .gt. 1) then
      call MPI_REDUCE(t, tmax, 1, MPI_REAL8, MPI_MAX, 0, &
                      MPI_COMM_WORLD, ierror)
   else
      tmax = t
   endif
   if (rank .eq. 0) then
      tavg = tavg + tmax
      t = t + flush(2)
      write(*,"(A, I3, A, I4, A, E9.3, A)") "On trial", k, &
            ", convergence after", it, " iterations in ", &
            tmax, " seconds."
   endif


   if (k .eq. ntrials) then
      if (rank .eq. 0) then
         tavg = 0.2*tavg
         write (*,"(A, E9.3, A)") "The average time was ", tavg, &
                                  " seconds."
      endif
      call OUTPUT(diamx, diamy, h, coord, nodes, gl_nbnodes, &
                  elem, elements, u, localglobal, rank, p)
   endif


   deallocate(u, rhs)
   deallocate(coord,elem,localglobal)
enddo

call mpi_finalize(ierror)

contains


subroutine INPUT(h, prec, coarse, overlap, exact, M, p, rank)
implicit none
integer, intent(in):: p, rank
integer, intent(out):: prec, coarse, overlap, exact, M
real*8, intent(out):: h

if (p .eq. 1) then
   print *, '1 processor'
   print *, 'Choose a mesh parameter h:'
```

```
      read(*,*) h
      print *, 'Standard (= 0) or preconditioned CG algorithm (= 1)?'
      read(*,*) prec
      if (prec .eq. 1) then
         print *,'Number of subdomains?'
         read(*,*) M
      endif
      overlap = 2
      exact = 1
      coarse = 0
   else
      if (rank == 0) then
         print *, p, ' processors'
         print *, 'Choose a mesh parameter h:'
         read(*,*) h
         print *, 'Standard (= 0) or preconditioned CG algorithm (= 1)?'
         read(*,*) prec
         if (prec .eq. 1) then
            print *, 'Coarse grid correction?'
            read(*,*) coarse
            if (coarse .eq. 0) then
               print *, 'How many layers of overlap for the decomposition?'
               read(*,*) overlap
            else
               overlap = 2
            endif
            print *, 'Exact (= 1) or inexact (= 0) solves on subdomains?'
            read(*,*) exact
         else
            overlap = 2
            coarse = 0
            exact = 0
         endif
      endif
      call mpi_bcast(h, 1, MPI_REAL8, 0, MPI_COMM_WORLD, ierror)
      call mpi_bcast(overlap, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierror)
      call mpi_bcast(coarse, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierror)
      call mpi_bcast(prec, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierror)
      call mpi_bcast(exact, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierror)
      M = p
   endif
   end subroutine INPUT


! Assemble the linear system:
   subroutine ASSEMBLE(coord, nodes, intnodes, gl_nbnodes,elem, elements,&
                       h, localglobal, xmin, diamy, rank, p, rhs, Stiff)
   implicit none
   integer, intent(in):: nodes, intnodes, gl_nbnodes, elements, rank, p
   integer, intent(in):: localglobal(nodes), elem(elements,3)
   real*8, intent(in):: h, xmin, diamy, coord(nodes,2)
   real*8, intent(out):: rhs(intnodes)
   type(sparsematrix), intent(out):: Stiff
   integer:: i, k
   integer:: v(3), loc_nbnodes, nvert, nhoriz
   integer:: is_i_internal, idxi
   real*8:: area, a(3), b(3), c(3), x(3), y(3)
```

```
    loc_nbnodes = nodes - intnodes
    nvert = NINT((diamy - h)/h)
    nhoriz = (loc_nbnodes - 2*nvert)/2
! calculate the right hand side:
    rhs(:) = 0.0
    do k = 1, elements
       v(1) = elem(k,1)
       v(2) = elem(k,2)
       v(3) = elem(k,3)
       x(1) = coord(v(1),1)
       y(1) = coord(v(1),2)
       x(2) = coord(v(2),1)
       y(2) = coord(v(2),2)
       x(3) = coord(v(3),1)
       y(3) = coord(v(3),2)
       area = 0.5*((x(1)*y(2) - x(2)*y(1)) + (x(3)*y(1) - x(1)*y(3)) &
                   + (x(2)*y(3) - x(3)*y(2)))
       if (localglobal(v(1)) .gt. gl_nbnodes) then
          a(1) = x(2)*y(3) - x(3)*y(2)
          b(1) = y(2) - y(3)
          c(1) = x(3) - x(2)
       else
          a(1) = 0
          b(1) = 0
          c(1) = 0
       endif
       if (localglobal(v(2)) .gt. gl_nbnodes) then
          a(2) = x(3)*y(1) - x(1)*y(3)
          b(2) = y(3) - y(1)
          c(2) = x(1) - x(3)
       else
          a(2) = 0
          b(2) = 0
          c(2) = 0
       endif
       if (localglobal(v(3)) .gt. gl_nbnodes) then
          a(3) = x(1)*y(2) - x(2)*y(1)
          b(3) = y(1) - y(2)
          c(3) = x(2) - x(1)
       else
          a(3) = 0
          b(3) = 0
          c(3) = 0
       endif
       do i = 1, 3
          is_i_internal = localglobal(v(i)) - gl_nbnodes
          if ((is_i_internal .gt. 0) .and. (v(i) .gt. loc_nbnodes)) then
             if (rank .eq. 0) then
                idxi = is_i_internal
             else
                idxi = NINT(is_i_internal - nvert*xmin/h)
             endif
             rhs(idxi) = rhs(idxi) + INTEGRAL(x,y,area,a,b,c,i)
          endif
       enddo
    enddo
```

```
! setup the stiffness matrix:
    call STIFF_SETUP(Stiff, intnodes, nvert)
    return
    end subroutine ASSEMBLE


    subroutine PARALLEL_PCG(Stiff, intnodes, rhs, coord, nodes, overlap, &
                            exact, coarse, omega, gl_nbnodes, u, &
                            h, it, diamx, diamy, rank, p)
    implicit none
    integer, intent(in):: intnodes, nodes, gl_nbnodes, rank, p
    integer, intent(in):: coarse, overlap, exact
    real*8, intent(in):: diamx, diamy, h, omega
    real*8, intent(in):: coord(nodes,2), rhs(intnodes)
    type(sparsematrix), intent(in) :: Stiff
    integer, intent(out):: it
    real*8, intent(out):: u(nodes)
    integer:: dim, dimadd, dim_dp, nvert, nhoriz, loc_nbnodes
    integer:: nodes_c, intnodes_c, nv_c, n
    integer:: i, j, idx1, idx2, idx3, idx4, idx5, idx6, idx7, stride
    real*8:: rz, rznew, qAq, alpha, beta, gl_rz, gl_rznew, gl_qAq
    real*8:: initerr, error, h_c
    real*8:: r(intnodes), Aq(intnodes), z(intnodes)
    real*8:: x0vec(intnodes), xvec(intnodes), ip(intnodes)
    real*8, allocatable:: uint(:), q(:), qadd1(:), qadd2(:), temp(:)
    real*8, allocatable:: r_c(:), z_c(:), x0vec_c(:), Rest_T(:,:)
    real*8, pointer:: coord_c(:,:), Rest(:,:)
    type(sparsematrix) :: A_bound, Stiff_c

! allocate the vectors uint and q and compute some indices:
    loc_nbnodes = nodes - intnodes
    nvert = NINT((diamy - h)/h)
    nhoriz = (loc_nbnodes - 2*nvert)/2
    if (rank .eq. 0 .or. rank .eq. p-1) then
       dim = intnodes + nvert
    else
       dim = intnodes + 2*nvert
    endif
    allocate(uint(dim), q(dim))
    dimadd = overlap*nvert
    allocate(qadd1(dimadd),qadd2(dimadd))
    if (rank .lt. p-1) then
       dim_dp = intnodes - (overlap - 1)*nvert
    else
       dim_dp = intnodes
    endif
    if (rank .eq. 0) then
       idx1 = 0
       idx7 = 0
    else
       idx1 = nvert
       idx7 = overlap/2*nvert
    endif
    idx2 = intnodes - overlap*nvert
    idx3 = (overlap - 1)*nvert
    idx4 = dim - (overlap + 1)*nvert
    idx5 = overlap*nvert
```

```
    idx6 = intnodes - (overlap - 1)*nvert
    stride = NINT(diamx/p/h)
! A_bound connects the nodes to the artificial boundary:
    call S_INIT(A_bound, 2*nvert, nvert)
    do i = 1, nvert
       A_bound%C(i) = -1.0
    enddo
! setup the coarse system:
    if (coarse .eq. 1) then
       call SETUP_COARSE(exact, h, diamx, diamy, rank, p, h_c, nodes_c, nv_c, &
                         Stiff_c, intnodes_c, coord, nodes, intnodes, Rest_T)
       allocate(r_c(intnodes_c), z_c(intnodes_c), temp(intnodes_c))
       allocate(x0vec_c(intnodes_c), Rest(intnodes_c,intnodes))
       call TRANSPOSE(Rest_T, intnodes, intnodes_c, Rest)
    endif

! Execute the steps of Algorithm 3.2:
! 1. start with an initial guess uint:
    uint(:) = 0.1
    x0vec = 0.1
    x0vec_c = 0.1

! 2. calculate the residual r and its infinity norm:
    r =  S_MATVEC(Stiff, uint(idx1+1:idx1+intnodes), intnodes)
    if (rank .lt. p-1) then
       r(intnodes-nvert+1:intnodes) = r(intnodes-nvert+1:intnodes) &
                           + S_MATVEC(A_bound, uint(dim-nvert+1:dim), nvert)
    endif
    if (rank .gt. 0) then
       r(1:nvert) = r(1:nvert) &
                  + S_MATVEC(A_bound, uint(1:nvert), nvert)
    endif
    r = rhs - r
    call MPI_ALLREDUCE(maxval(abs(r)), initerr, 1, MPI_REAL8, MPI_MAX, &
                       MPI_COMM_WORLD, ierror)

! 3. compute the first correction z and direction q:
    if (exact .eq. 1) then
       call PCG(Stiff, r, x0vec, z, 1000, nvert, omega)
       x0vec = z
    else
       call APPLY_SSOR(r, z, intnodes, nvert, omega)
    endif
    if (coarse .eq. 1) then
       temp = matmul(Rest(:,idx7+1:intnodes), r(idx7+1:intnodes))
       call MPI_ALLREDUCE(temp, r_c, intnodes_c, MPI_REAL8, MPI_SUM, &
                          MPI_COMM_WORLD, ierror)
       call PCG(Stiff_c, r_c, x0vec_c, z_c, 1000, nv_c, omega)
       x0vec_c = z_c
       ip = matmul(Rest_T, z_c)
    endif
    if (rank .lt. p-1) then
       if (coarse .eq. 1) then
          z(idx2+1:idx2+nvert) = z(idx2+1:idx2+nvert) + ip(idx2+1:idx2+nvert)
       endif
       call MPI_SEND(z(idx2+1:intnodes), dimadd, MPI_REAL8, &
                     rank+1, 1, MPI_COMM_WORLD, ierror)
```

```
    endif
    if(rank .gt. 0) then
       call MPI_RECV(qadd1(1), dimadd, MPI_REAL8, rank-1, 1, &
                        MPI_COMM_WORLD, status, ierror)
       q(1:nvert) = qadd1(1:nvert)
       if (coarse .eq. 1) then
          z(dimadd-nvert+1:dimadd) = z(dimadd-nvert+1:dimadd) &
                                   + ip(dimadd-nvert+1:dimadd)
       endif
       call MPI_SEND(z(1), dimadd, MPI_REAL8, rank-1, 2, &
                        MPI_COMM_WORLD, ierror)
       z(1:dimadd-nvert) = z(1:dimadd-nvert) + qadd1(nvert+1:dimadd)
    endif
    if(rank .lt. p-1) then
       call MPI_RECV(qadd2(1), dimadd, MPI_REAL8, rank+1, 2, &
                        MPI_COMM_WORLD, status, ierror)
       q(dim-nvert+1:dim) = qadd2(dimadd-nvert+1:dimadd)
       z(intnodes-dimadd+nvert+1:intnodes) &
           = z(intnodes-dimadd+nvert+1:intnodes) + qadd2(1:dimadd-nvert)
    endif
    if (coarse .eq. 1) then
       if (rank .eq. 0) then
          z(1:idx2) = z(1:idx2) + ip(1:idx2)
          z(idx2+nvert+1:intnodes) = z(idx2+nvert+1:intnodes) &
                                   + ip(idx2+nvert+1:intnodes)
       else if (rank .eq. p-1) then
          z(1:dimadd-nvert) = z(1:dimadd-nvert) + ip(1:dimadd-nvert)
          z(dimadd+1:intnodes) = z(dimadd+1:intnodes) + ip(dimadd+1:intnodes)
       else
          z(1:idx3) = z(1:idx3) + ip(1:idx3)
          z(dimadd+1:intnodes-dimadd) = z(dimadd+1:intnodes-dimadd) &
                                      + ip(dimadd+1:intnodes-dimadd)
          z(intnodes-idx3+1:intnodes) = z(intnodes-idx3+1:intnodes) &
                                      + ip(intnodes-idx3+1:intnodes)
       endif
    endif
    q(idx1+1:idx1+intnodes) = z

! 4a. Calculate the dotproduct (r,z):
    rz = DOTPRODUCT(r(1:dim_dp),z(1:dim_dp),dim_dp)
    call MPI_ALLREDUCE(rz, gl_rz, 1, MPI_REAL8, MPI_SUM, &
                       MPI_COMM_WORLD, ierror)

    do i = 1, 500

! 4b. Calculate qAq and alpha:
       Aq = S_MATVEC(Stiff, q(idx1+1:idx1+intnodes), intnodes)
       if (rank .lt. p-1) then
          Aq(intnodes-nvert+1:intnodes) = Aq(intnodes-nvert+1:intnodes) &
                        + S_MATVEC(A_bound, q(dim-nvert+1:dim), nvert)
       endif
       if (rank .gt. 0) then
          Aq(1:nvert) = Aq(1:nvert) + S_MATVEC(A_bound, q(1:nvert), nvert)
       endif
       qAq = DOTPRODUCT(q(idx1+1:idx1+dim_dp), Aq(1:dim_dp),dim_dp)
       call MPI_ALLREDUCE(qAq, gl_qAq, 1, MPI_REAL8, MPI_SUM, &
                          MPI_COMM_WORLD, ierror)
```

```
      alpha = gl_rz/gl_qAq

! 5. Compute the new solution iterate uint:
      uint = uint + alpha*q

! 6. Calculate the residual r and check for convergence:
      r = r - alpha*Aq
      call MPI_ALLREDUCE(maxval(abs(r)), error, 1, MPI_REAL8, MPI_MAX, &
                         MPI_COMM_WORLD, ierror)
      if (error/initerr < 0.001) then
         it = i
         goto 87
      endif

! 7. Compute the correction z:
      if (exact .eq. 1) then
         z = r
         call PCG(Stiff, z, x0vec, xvec, 1000, nvert, omega)
         z = xvec
         x0vec = z
      else
         call APPLY_SSOR(r, z, intnodes, nvert, omega)
      endif
      if (rank .lt. p-1) then
         call MPI_SEND(z(idx6+1:intnodes), idx3, MPI_REAL8, &
                       rank+1, 1, MPI_COMM_WORLD, ierror)
      endif
      if(rank .gt. 0) then
         call MPI_RECV(qadd1(1), idx3, MPI_REAL8, rank-1, 1, &
                          MPI_COMM_WORLD, status, ierror)

         call MPI_SEND(z(1:idx3), idx3, MPI_REAL8, rank-1, 2, &
                          MPI_COMM_WORLD, ierror)
         z(1:idx3) = z(1:idx3) + qadd1(1:idx3)
      endif
      if(rank .lt. p-1) then
         call MPI_RECV(qadd2(1), idx3, MPI_REAL8, rank+1, 2, &
                          MPI_COMM_WORLD, status, ierror)
         z(intnodes-idx3+1:intnodes) &
               = z(intnodes-idx3+1:intnodes) + qadd2(1:idx3)
      endif
      if (coarse .eq. 1) then
         temp = matmul(Rest(:,idx7+1:intnodes), r(idx7+1:intnodes))
         call MPI_ALLREDUCE(temp, r_c, intnodes_c, MPI_REAL8, MPI_SUM, &
                            MPI_COMM_WORLD, ierror)
         call PCG(Stiff_c, r_c, x0vec_c, z_c, 1000, nv_c, omega)
         x0vec_c = z_c
         ip = matmul(Rest_T, z_c)
         z = z + ip
      endif

! 8. and 4a. Calculate the new dotproduct (r,z) and beta:
      rznew = DOTPRODUCT(r(1:dim_dp),z(1:dim_dp),dim_dp)
      call MPI_ALLREDUCE(rznew, gl_rznew, 1, MPI_REAL8, MPI_SUM, &
                         MPI_COMM_WORLD, ierror)
      beta = gl_rznew/gl_rz
      gl_rz = gl_rznew
```

```
! 9. Compute the direction q:
      q(idx1+1:idx1+intnodes) = z + beta*q(idx1+1:idx1+intnodes)
      if (rank .lt. p-1) then
         call MPI_SEND(q(idx4+1), nvert, MPI_REAL8, rank+1, 3, &
                       MPI_COMM_WORLD, ierror)
      endif
      if(rank .gt. 0) then
         call MPI_RECV(q(1:nvert), nvert, MPI_REAL8, rank-1, 3, &
                       MPI_COMM_WORLD, status, ierror)
         call MPI_SEND(q(idx5+1), nvert, MPI_REAL8, rank-1, 4, &
                       MPI_COMM_WORLD, ierror)
      endif
      if(rank .lt. p-1) then
         call MPI_RECV(q(dim-nvert+1), nvert, MPI_REAL8, rank+1, 4, &
                       MPI_COMM_WORLD, status, ierror)
      endif
   enddo

   print *,'Did not converge.'

87 deallocate(qadd1,qadd2)
   if (coarse .eq. 1) then
      deallocate(r_c, z_c, temp, x0vec_c, Rest, Rest_T)
   endif

! Include the boundary values to get the solution u:
   do i = 1, nodes
      u(i) = u0(coord(i,1),coord(i,2))
   enddo
   if (rank .eq. 0) then
      u(loc_nbnodes+1:nodes) = u(loc_nbnodes+1:nodes) + uint(1:intnodes)
      u(nhoriz+1:nhoriz+nvert) = u(nhoriz+1:nhoriz+nvert) &
                               + uint(intnodes+1:dim)
   else if (rank .eq. p-1) then
      u(2*nhoriz+nvert+1:nodes) = u(2*nhoriz+nvert+1:nodes) + uint
   else
      u(2*nhoriz+nvert+1:nodes) = u(2*nhoriz+nvert+1:nodes) &
                                + uint(1:intnodes+nvert)
      u(nhoriz+1:nhoriz+nvert) = u(nhoriz+1:nhoriz+nvert) &
                               + uint(dim-nvert+1:dim)
   endif
   deallocate(uint)
   return
   end subroutine PARALLEL_PCG



   subroutine PARALLEL_CG(Stiff, intnodes, rhs, nodes, overlap, &
                          gl_nbnodes, u, h, it, diamx, diamy, rank, p)
   implicit none
   integer, intent(in):: intnodes, nodes, gl_nbnodes, rank, p
   integer, intent(in):: overlap
   real*8, intent(in):: rhs(intnodes), diamx, diamy, h
   type(sparsematrix), intent(in) :: Stiff
   integer, intent(out):: it
   real*8, intent(out):: u(nodes)
```

```
      integer:: i, j, nhoriz, nvert, idx1, idx2, idx4, idx5, dimadd, dim_dp
      integer:: n, loc_nbnodes, dim
      type(sparsematrix) :: A_bound
      real*8:: initerr, error
      real*8:: rz, rznew, qAq, gl_rz, gl_rznew, gl_qAq
      real*8:: alpha, beta
      real*8:: x0vec(intnodes)
      real*8:: r(intnodes), Aq(intnodes), z(intnodes)
      real*8, allocatable:: uint(:), q(:)

      loc_nbnodes = nodes - intnodes
      nvert = NINT((diamy - h)/h)
      nhoriz = (loc_nbnodes - 2*nvert)/2
      if (rank .eq. 0 .or. rank .eq. p-1) then
         dim = intnodes + nvert
      else
         dim = intnodes + 2*nvert
      endif
      allocate(uint(dim), q(dim))
      if (rank .eq. 0) then
         idx1 = 0
      else
         idx1 = nvert
      endif
      idx2 = intnodes - overlap*nvert
      idx4 = dim - (overlap + 1)*nvert
      idx5 = overlap*nvert

!     A_bound connects the nodes to the artificial boundary:
      call S_INIT(A_bound, 2*nvert, nvert)
      do i = 1, nvert
         A_bound%C(i) = -1.0
      enddo

      dimadd = overlap*nvert
      if (rank .lt. p-1) then
         dim_dp = intnodes - (overlap - 1)*nvert
      else
         dim_dp = intnodes
      endif

!     iterate:
      uint(:) = 0.1
      x0vec = 0.1
      r =  S_MATVEC(Stiff, uint(idx1+1:idx1+intnodes), intnodes)
      if (rank .lt. p-1) then
         r(intnodes-nvert+1:intnodes) = r(intnodes-nvert+1:intnodes) &
                              + S_MATVEC(A_bound, uint(dim-nvert+1:dim), nvert)
      endif
      if (rank .gt. 0) then
         r(1:nvert) = r(1:nvert) &
                     + S_MATVEC(A_bound, uint(1:nvert), nvert)
      endif
      r = rhs - r

      call MPI_ALLREDUCE(maxval(abs(r)), initerr, 1, MPI_REAL8, MPI_MAX, &
                         MPI_COMM_WORLD, ierror)
```

```
   if (rank .lt. p-1) then
      call MPI_SEND(r(idx2+1:idx2+nvert), nvert, MPI_REAL8, &
                    rank+1, 1, MPI_COMM_WORLD, ierror)
   endif

   if(rank .gt. 0) then
      call MPI_RECV(q(1), nvert, MPI_REAL8, rank-1, 1, &
                       MPI_COMM_WORLD, status, ierror)
      call MPI_SEND(r(dimadd-nvert+1), nvert, MPI_REAL8, rank-1, 2, &
                       MPI_COMM_WORLD, ierror)
   endif
   if(rank .lt. p-1) then
      call MPI_RECV(q(dim-nvert+1), dimadd, MPI_REAL8, rank+1, 2, &
                       MPI_COMM_WORLD, status, ierror)
   endif
   q(idx1+1:idx1+intnodes) = r

   rz = DOTPRODUCT(r(1:dim_dp),r(1:dim_dp),dim_dp)
   call MPI_ALLREDUCE(rz, gl_rz, 1, MPI_REAL8, MPI_SUM, &
                      MPI_COMM_WORLD, ierror)

   do i = 1, 1000
      Aq = S_MATVEC(Stiff, q(idx1+1:idx1+intnodes), intnodes)
      if (rank .lt. p-1) then
         Aq(intnodes-nvert+1:intnodes) = Aq(intnodes-nvert+1:intnodes) &
                       + S_MATVEC(A_bound, q(dim-nvert+1:dim), nvert)
      endif
      if (rank .gt. 0) then
         Aq(1:nvert) = Aq(1:nvert) + S_MATVEC(A_bound, q(1:nvert), nvert)
      endif
      qAq = DOTPRODUCT(q(idx1+1:idx1+dim_dp), Aq(1:dim_dp),dim_dp)

      call MPI_ALLREDUCE(qAq, gl_qAq, 1, MPI_REAL8, MPI_SUM, &
                            MPI_COMM_WORLD, ierror)
      alpha = gl_rz/gl_qAq
      if (rank .eq. 0) then
!         print *, alpha
      endif
      uint = uint + alpha*q
      r = r - alpha*Aq
      call MPI_ALLREDUCE(maxval(abs(r)), error, 1, MPI_REAL8, MPI_MAX, &
                       MPI_COMM_WORLD, ierror)
      if (error/initerr < 0.001) then
         it = i
         goto 87
      endif
      rznew = DOTPRODUCT(r(1:dim_dp),r(1:dim_dp),dim_dp)
      call MPI_ALLREDUCE(rznew, gl_rznew, 1, MPI_REAL8, MPI_SUM, &
                       MPI_COMM_WORLD, ierror)
      beta = gl_rznew/gl_rz
      if (rank .eq. 0) then
!         print *, beta
      endif
      gl_rz = gl_rznew
      q(idx1+1:idx1+intnodes) = r + beta*q(idx1+1:idx1+intnodes)
      if (rank .lt. p-1) then
```

```
         call MPI_SEND(q(idx4+1), nvert, MPI_REAL8, rank+1, 3, &
                       MPI_COMM_WORLD, ierror)
      endif
      if(rank .gt. 0) then
         call MPI_RECV(q(1:nvert), nvert, MPI_REAL8, rank-1, 3, &
                       MPI_COMM_WORLD, status, ierror)
         call MPI_SEND(q(idx5+1), nvert, MPI_REAL8, rank-1, 4, &
                       MPI_COMM_WORLD, ierror)
      endif
      if(rank .lt. p-1) then
         call MPI_RECV(q(dim-nvert+1), nvert, MPI_REAL8, rank+1, 4, &
                       MPI_COMM_WORLD, status, ierror)
      endif
   enddo

   print *,'Did not converge.'
!   call printout(Stiff)

87 do i = 1, nodes
      u(i) = u0(coord(i,1),coord(i,2))
   enddo
   if (rank .eq. 0) then
      u(loc_nbnodes+1:nodes) = u(loc_nbnodes+1:nodes) + uint(1:intnodes)
      u(nhoriz+1:nhoriz+nvert) = u(nhoriz+1:nhoriz+nvert) &
                               + uint(intnodes+1:dim)
   else if (rank .eq. p-1) then
      u(2*nhoriz+nvert+1:nodes) = u(2*nhoriz+nvert+1:nodes) + uint
   else
      u(2*nhoriz+nvert+1:nodes) = u(2*nhoriz+nvert+1:nodes) &
                                + uint(1:intnodes+nvert)
      u(nhoriz+1:nhoriz+nvert) = u(nhoriz+1:nhoriz+nvert) &
                               + uint(dim-nvert+1:dim)
   endif
   deallocate(uint)
   return
   end subroutine PARALLEL_CG


   subroutine SOLVE_SERIAL(Stiff, intnodes, rhs, nodes, overlap, &
                           u, h, it, diamx, diamy, omega, M)
   implicit none
   integer, intent(in):: intnodes, nodes, overlap, M
   real*8, intent(in):: rhs(intnodes), diamx, diamy, h, omega
   type(sparsematrix), intent(in) :: Stiff
   integer, intent(out):: it
   real*8, intent(out):: u(nodes)
   integer:: i, j, nvert, dim_l, dim_r, dim_m
   real*8:: t, initerr, error
   real*8:: rz, rznew, qAq
   real*8:: uint(intnodes), q(intnodes), alpha, beta
   real*8:: x0vec(intnodes)
   real*8:: r(intnodes), Aq(intnodes), z(intnodes), zadd(intnodes)
   type(sparsematrix):: Stiff_l, Stiff_r, Stiff_m

   nvert = NINT((diamy - h)/h)
   if (MOD(overlap,2) .eq. 0) then
      dim_l = nvert*(NINT(diamx/M/h) - 1 + overlap/2)
```

```
      dim_r = dim_l
      dim_m = dim_l + nvert*(overlap/2)
   else
      dim_l = nvert*(NINT(diamx/M/h) + overlap/2)
      dim_r = dim_l - nvert
      dim_m = dim_r + nvert*(overlap/2)
   endif
   call STIFF_SETUP(Stiff_l, dim_l, nvert)
   call STIFF_SETUP(Stiff_m, dim_m, nvert)
   call STIFF_SETUP(Stiff_r, dim_r, nvert)

!  iterate:
   uint(:) = 0.1

   r = rhs - S_MATVEC(Stiff, uint, intnodes)
   initerr = maxval(abs(r))

   x0vec = 0.1
   z = 0.0
   zadd = 0.0
   call PCG(Stiff_l, r(1:dim_l), x0vec(1:dim_l), z(1:dim_l), 1000, nvert, omega)
   x0vec(1:dim_l) = z(1:dim_l)
   i = dim_l - (overlap - 1)*nvert
   do j = 2, M-1
      call PCG(Stiff_m, r(i+1:i+dim_m), x0vec(i+1:i+dim_m), &
              zadd(i+1:i+dim_m), 1000, nvert, omega)
      x0vec(i+1:i+dim_m) = zadd(i+1:i+dim_m)
      z(i+1:i+dim_m) = z(i+1:i+dim_m) + zadd(i+1:i+dim_m)
      i = i + dim_m - (overlap - 1)*nvert
   enddo
   call PCG(Stiff_r, r(i+1:i+dim_r), x0vec(i+1:i+dim_r), &
           zadd(i+1:i+dim_r), 1000, nvert, omega)
   x0vec(i+1:i+dim_r) = zadd(i+1:i+dim_r)
   z(i+1:i+dim_r) = z(i+1:i+dim_r) + zadd(i+1:i+dim_r)
   x0vec = z
   q = z

   rz = DOTPRODUCT(r, z, intnodes)

   do it = 1, 500

      Aq = S_MATVEC(Stiff, q, intnodes)
      qAq = DOTPRODUCT(q, Aq, intnodes)
      alpha = rz/qAq

      uint = uint + alpha*q

      r = r - alpha*Aq
      error = maxval(abs(r))
      if (error/initerr < 0.001) then
         goto 87
      endif

      z = 0.0
      zadd = 0.0
      call PCG(Stiff_l, r(1:dim_l), x0vec(1:dim_l), z(1:dim_l), 1000, nvert, omega)
      i = dim_l - (overlap - 1)*nvert
```

```
      do j = 2, M-1
         call PCG(Stiff_m, r(i+1:i+dim_m), x0vec(i+1:i+dim_m), &
                  zadd(i+1:i+dim_m), 1000, nvert, omega)
         z(i+1:i+dim_m) = z(i+1:i+dim_m) + zadd(i+1:i+dim_m)
         i = i + dim_m - (overlap - 1)*nvert
      enddo
      call PCG(Stiff_r, r(i+1:i+dim_r), x0vec(i+1:i+dim_r), &
               zadd(i+1:i+dim_r), 1000, nvert, omega)
      z(i+1:i+dim_r) = z(i+1:i+dim_r) + zadd(i+1:i+dim_r)
      x0vec = z

      rznew = DOTPRODUCT(r, z, intnodes)
      beta = rznew/rz
      rz = rznew

      q = z + beta*q
   enddo
   print *,'Did not converge.'

87 do i = 1, nodes
      u(i) = u0(coord(i,1),coord(i,2))
   enddo
   u(nodes-intnodes+1:nodes) = u(nodes-intnodes+1:nodes) + uint
   return
   end subroutine SOLVE_SERIAL


! Setup the coarse stiffness matrix and the interpolation matrix:
   subroutine SETUP_COARSE(exact, h, diamx, diamy, rank, p, h_c, nodes_c, nv_c, &
                           Stiff_c, intnodes_c, coord, nodes, intnodes, R_T)
   implicit none
   integer, intent(in):: exact, nodes, intnodes, p, rank
   real*8, intent(in):: diamx, diamy, h, coord(nodes,2)
   integer, intent(out):: nodes_c, intnodes_c, nv_c
   real*8, intent(out):: h_c
   real*8, allocatable, intent(out):: R_T(:,:)
   type(sparsematrix), intent(out):: Stiff_c
   integer:: elements_c, nh_c, nbnodes_c
   integer:: loc_nbnodes, nhoriz, nvert, n, pby8
   integer:: i, j, k, m, idx1, idx2, idx3, idx4, idx5, idx6
   integer, pointer:: elem_c(:,:)
   real*8:: a, b, x, y, lx, rx, ly, uy
   real*8, pointer:: coord_c(:,:)

   if (p .lt. 9 .or. exact .eq. 1) then
      h_c = diamx/p
   else
      h_c = diamx/8
   endif

! Get the coarse mesh:
   nh_c = NINT(diamx/h_c) + 1
   nv_c = NINT((diamy - h_c)/h_c) + 2
   nbnodes_c = 2*(nh_c + nv_c) - 4
   nodes_c = nh_c*nv_c
   intnodes_c = nodes_c - nbnodes_c
   elements_c = 2*(nh_c - 1)*(nv_c - 1)
```

```
      call MESH_RECTANGLE(0.0, diamx, diamy, h_c, coord_c, nodes_c, &
                          elem_c, elements_c)
      nv_c = nv_c - 2
      nh_c = nh_c - 2

! Calculate the stiffness matrix:
      call STIFF_SETUP(Stiff_c, intnodes_c, nv_c)

! Compute the interpolation matrix R_T:
      allocate(R_T(intnodes, intnodes_c))
      R_T = 0.0
      nvert = NINT(diamy/h) - 1
      nhoriz = intnodes/nvert
      n = NINT(h_c/h)
      loc_nbnodes = 2*(nhoriz + nvert) + 4
      nbnodes_c = nodes_c - intnodes_c
      pby8 = p/8
      do i = 1, nhoriz
         if (p .lt. 9 .or. exact .eq. 1) then
            if (rank .eq. 0) then
               idx1 = 1
               idx2 = 2
               idx3 = nbnodes_c + 1
               idx4 = nbnodes_c
            else if (rank .eq. p-1) then
               idx1 = p
               idx2 = p + 1
               idx3 = p + 2
               idx4 = nodes_c - nv_c + 1
            else
               idx1 = rank + 1
               idx2 = rank + 2
               idx3 = nbnodes_c + rank*nv_c + 1
               idx4 = nbnodes_c + (rank - 1)*nv_c + 1
            endif
         else
            if (rank .lt. pby8) then
               idx1 = 1
               idx2 = 2
               idx3 = nbnodes_c + 1
               idx4 = nbnodes_c
            else if (rank .ge. 7*pby8) then
               idx1 = nh_c + 1
               idx2 = nh_c + 2
               idx3 = nh_c + 3
               idx4 = nodes_c - nv_c + 1
            else
               idx1 = rank/pby8 + 1
               idx2 = rank/pby8 + 2
               idx3 = nbnodes_c + int(rank/pby8)*nv_c + 1
               idx4 = idx3 - nv_c
            endif
         endif
         do j = 1, nv_c + 1
            lx = coord_c(idx1,1)
            rx = coord_c(idx2,1)
            ly = coord_c(idx1,2)
```

```
      uy = coord_c(idx4,2)
      idx5 = (i - 1)*nvert + (j - 1)*n + 1
      if (j .eq. nv_c + 1) then
         idx6 = idx5 + n - 2
      else
         idx6 = idx5 + n - 1
      endif
      do k = idx5, idx6
         x = coord(k + loc_nbnodes,1)
         y = coord(k + loc_nbnodes,2)
         a = (x - lx)/(rx - lx)
         b = (y - ly)/(uy - ly)
         if (idx1 .gt. nbnodes_c) then
            R_T(k,idx1-nbnodes_c) = (1 - a)*(1 - b)
         endif
         if (idx2 .gt. nbnodes_c) then
            R_T(k,idx2-nbnodes_c) = a*(1 - b)
         endif
         if (idx3 .gt. nbnodes_c) then
            R_T(k,idx3-nbnodes_c) = a*b
         endif
         if (idx4 .gt. nbnodes_c) then
            R_T(k,idx4-nbnodes_c) = (1 - a)*b
         endif
      enddo
      idx1 = idx4
      idx2 = idx3
      if (j .ne. nv_c) then
         idx3 = idx3 + 1
         if ((p .lt. 9 .and. rank .eq.0) .or. &
             (p .ge. 9 .and. rank .lt. pby8)) then
            idx4 = idx4 - 1
         else
            idx4 = idx4 + 1
         endif
      else
         if (p .lt. 9 .or. exact .eq. 1) then
            idx3 = nbnodes_c - nv_c - rank - 1
         else
            idx3 = nbnodes_c - nv_c - rank/pby8 - 1
         endif
         idx4 = idx3 + 1
      endif
   enddo
enddo
return
end subroutine SETUP_COARSE


real*8 function INTEGRAL(x,y,area,a,b,c,i)
implicit none
real*8, intent(in):: area, a(3), b(3), c(3), x(3), y(3)
integer, intent(in):: i
real*8:: zx(4), zy(4), w(4), xi(4), eta(4), chi(4)
integer:: j
xi(1) = 1.0/3.0
xi(2) = 0.2
```

```
xi(3) = 0.6
xi(4) = 0.2
eta(1) = 1.0/3.0
eta(2) = 0.2
eta(3) = 0.2
eta(4) = 0.6
chi(1) = 1.0/3.0
chi(2) = 0.6
chi(3) = 0.2
chi(4) = 0.2
w(1) = -27.0/48.0
w(2) = 25.0/48.0
w(3) = w(2)
w(4) = w(3)
INTEGRAL = 0.0
do j = 1, 4
   zx(j) = xi(j)*x(1) + eta(j)*x(2) + chi(j)*x(3)
   zy(j) = xi(j)*y(1) + eta(j)*y(2) + chi(j)*y(3)
   INTEGRAL = INTEGRAL + (f(zx(j),zy(j)) - Lu0(zx(j),zy(j))) &
                         *(a(i) + b(i)*zx(j) + c(i)*zy(j))*w(j)
enddo
INTEGRAL = 0.5*INTEGRAL
return
end function INTEGRAL

end program PARALLEL_ADDITIVE_SCHWARZ
```

B.2 meshmod.f90

```
module MESH_MODULE

include "mpif.h"

contains

subroutine DECOMPOSE_AND_MESH(diamx, diamy, h, overlap, coord, nodes, &
    localglobal, elem, elements, gl_nbnodes, intnodes, xmin, xmax, p, rank)
implicit none
real*8, intent(in):: diamx, diamy, h
integer, intent(in):: overlap, p, rank
integer, intent(out):: nodes, elements, gl_nbnodes, intnodes
real*8, intent(out):: xmin, xmax
integer:: i, nvert, nhoriz, loc_nbnodes, nod
real*8, pointer, dimension(:,:):: coord
integer, pointer, dimension(:,:):: elem
integer, pointer, dimension(:):: localglobal

nvert = NINT((diamy - h)/h) + 2
if (p .eq. 1) then
   nhoriz = NINT(diamx/h) + 1
   xmin = 0.0
   xmax = diamx
else if (rank .eq. 0) then
   if (MOD(overlap,2) .eq. 0) then
      nhoriz = NINT((diamx/p)/h) + 1 + overlap/2
      xmin = 0.0
      xmax = diamx/p + overlap/2*h
```

```
      else
         nhoriz = NINT((diamx/p)/h) + 2 + overlap/2
         xmin = 0.0
         xmax = diamx/p + (overlap + 1)/2*h
      endif
   else if (rank .eq. p-1) then
      nhoriz = NINT((diamx/p)/h) + 1 + overlap/2
      xmin = diamx - diamx/p - (overlap/2)*h
      xmax = diamx
   else
      if (MOD(overlap,2) .eq. 0) then
         nhoriz = NINT(diamx/p/h) + 1 + overlap
         xmin = rank*(diamx/p) - overlap/2*h
         xmax = (rank + 1)*(diamx/p) + overlap/2*h
      else
         nhoriz = NINT(diamx/p/h) + 1 + overlap
         xmin = rank*(diamx/p) - (overlap/2)*h
         xmax = (rank + 1)*(diamx/p) + (overlap + 1)/2*h
      endif
   endif
   loc_nbnodes = 2*(nhoriz + nvert) - 4
   nodes = nhoriz*nvert
   intnodes = (nhoriz - 2)*(nvert - 2)
   gl_nbnodes = NINT(2*(diamx + diamy)/h)
   elements = 2*(nhoriz - 1)*(nvert - 1)
   allocate(localglobal(nodes))

!  setup coord and elem:
   call MESH_RECTANGLE(xmin, xmax, diamy, h, coord, nodes, elem, &
                       elements)

!  setup the vector localglobal:
   do i = 1, nodes
      if (rank .eq. 0) then
         if (i .gt. nhoriz) then
            if (i .lt. nhoriz + nvert - 1) then
               localglobal(i) = i - nhoriz + gl_nbnodes + intnodes
            else
               localglobal(i) = NINT(i - nhoriz - nvert + 2 &
                                + (2*diamx + diamy - xmax)/h)
            endif
         else
            localglobal(i) = i
         endif
      else if (rank .eq. p-1) then
         if (i .le. 2*nhoriz + nvert - 2) then
            localglobal(i) = NINT(i + xmin/h)
         else if (i .gt. loc_nbnodes) then
            localglobal(i) = NINT(i - loc_nbnodes + gl_nbnodes  &
                             + xmin*(nvert - 2)/h)
         else
            nod = loc_nbnodes + 2*nhoriz + nvert  - i - 1
            localglobal(i) = NINT(nod - 2*nhoriz - nvert + 2 + gl_nbnodes &
                             + (xmin - h)*(nvert - 2)/h)
         endif
      else
         if (i .le. nhoriz) then
```

```
         localglobal(i) = NINT(i + xmin/h)
      else if(i .ge. (nhoriz + nvert - 1) &
            .and. i .le. (2*nhoriz + nvert - 2)) then
         localglobal(i) = NINT(i - nhoriz - nvert + 2 &
                        + (2*diamx + diamy - xmax)/h)
      else if(i .lt. (nhoriz + nvert - 1)) then
         localglobal(i) = NINT(i - nhoriz + gl_nbnodes &
                              + (xmax - h)*(nvert - 2)/h)
      else if(i .gt. loc_nbnodes) then
         localglobal(i) = int(i - loc_nbnodes + gl_nbnodes &
                        + xmin*(nvert - 2)/h)
      else
         nod = loc_nbnodes + 2*nhoriz + nvert - i - 1
         localglobal(i) = NINT(nod - 2*nhoriz - nvert + 2 + gl_nbnodes &
                        + (xmin - h)*(nvert - 2)/h)
      endif
   endif
enddo

end subroutine DECOMPOSE_AND_MESH


subroutine MESH_RECTANGLE(xmin, xmax, diamy, h, coord, nodes, elem, &
                          elements)
implicit none
real*8, intent(in):: xmin, xmax, diamy, h
integer, intent(out):: nodes, elements
integer, pointer:: elem(:,:)
real*8, pointer:: coord(:,:)
integer:: i, j, k, index, nhoriz, nvert, loc_nbnodes
real*8:: x, y

nvert = NINT((diamy - h)/h) + 2
nhoriz = NINT((xmax - xmin)/h) + 1
loc_nbnodes = 2*(nhoriz + nvert) - 4
nodes = nhoriz*nvert
elements = 2*(nhoriz - 1)*(nvert - 1)
allocate(coord(nodes,2),elem(elements,3))
x = xmin
j = loc_nbnodes - nvert + 2
k = 1
do i = 1, nhoriz
   coord(i,1) = x
   coord(i,2) = 0.0
   coord(j,1) = x
   coord(j,2) = diamy
   if (i .ne. nhoriz) then
      elem(k,1) = i
      elem(k,2) = i + 1
      if (i .eq. 1) then
         elem(k,3) = loc_nbnodes
      else
         elem(k,3) = loc_nbnodes + (i - 2)*(nvert - 2) + 1
      endif
      k = k + 1
      elem(k,1) = i + 1
      if (i .eq. nhoriz - 1) then
```

```
            elem(k,2) = nhoriz + 1
         else
            elem(k,2) = loc_nbnodes + (i - 1)*(nvert - 2) + 1
         endif
         if (i .eq. 1) then
            elem(k,3) = loc_nbnodes
         else
            elem(k,3) = loc_nbnodes + (i - 2)*(nvert - 2) + 1
         endif
         k = k + 1
         elem(k,1) = j - 1
         elem(k,2) = j
         if (i .eq. nhoriz - 1) then
            elem(k,3) = j - 2
         else
            elem(k,3) = loc_nbnodes + i*(nvert - 2)
         endif
         k = k + 1
         elem(k,1) = j
         if (i .eq. 1) then
            elem(k,2) = j + 1
         else
            elem(k,2) = loc_nbnodes + (i - 1)*(nvert - 2)
         endif
         if (i .eq. nhoriz - 1) then
            elem(k,3) = nhoriz + nvert - 2
         else
            elem(k,3) = loc_nbnodes + i*(nvert - 2)
         endif
         k = k + 1
      endif
      x = x + h
      j = j - 1
   enddo
   y = h
   j = loc_nbnodes
   do i = nhoriz + 1, nhoriz + nvert - 2
      coord(i,1) = xmax
      coord(i,2) = y
      coord(j,1) = xmin
      coord(j,2) = y
      if (i .ne. nhoriz + nvert - 2) then
         elem(k,1) = i
         elem(k,2) = i + 1
         elem(k,3) = nodes - nvert + 3 + i - nhoriz
         k = k + 1
         elem(k,1) = j - 1
         elem(k,2) = j
         elem(k,3) = loc_nbnodes + i - nhoriz
         k = k + 1
      endif
      y = y + h
      j = j - 1
   enddo
   x = xmin
   do i = 1, nhoriz - 2
      x = x + h
```

```
   y = 0.0
   do j = 1, nvert - 2
      y = y + h
      index = loc_nbnodes + (nvert - 2)*(i - 1) + j
      coord(index,1) = x
      coord(index,2) = y
      if (j .ne. nvert - 2) then
         elem(k,1) = index
         elem(k,2) = index + 1
         if (i .eq. 1) then
            elem(k,3) = loc_nbnodes - j
         else
            elem(k,3) = loc_nbnodes + (i - 2)*(nvert - 2) + j + 1
         endif
         k = k + 1
         elem(k,1) = index + 1
         elem(k,2) = index
         if (i .eq. nhoriz - 2) then
            elem(k,3) = nhoriz + j
         else
            elem(k,3) = loc_nbnodes + i*(nvert - 2) + j
         endif
         k = k + 1
      endif
   enddo
enddo
return
end subroutine MESH_RECTANGLE


subroutine OUTPUT(diamx, diamy, h, coord, nodes, gl_nbnodes, &
                  elem, elements, u, localglobal, rank, p)
implicit none
integer, intent(in):: nodes, gl_nbnodes, elements, rank, p
integer, intent(inout):: elem(elements,3), localglobal(nodes)
real*8, intent(in):: diamx, diamy, h, coord(nodes,2), u(nodes)
integer:: i, j, allnodes, allelements, sum, ierror
integer:: elemt(3,elements), segadd(gl_nbnodes,5), gl_seg(gl_nbnodes,2)
integer:: vecelements(p), disp(p), blen(p)
integer, allocatable:: elemadd(:), glob_elem(:,:)
real*8, allocatable:: lg_u(:), glob_u(:), lgcord(:,:), glob_cord(:,:)

if (p .gt. 1) then
   allnodes = NINT((diamx/h + 1)*(diamy/h + 1))
else
   allnodes = nodes
endif
allocate(lg_u(allnodes))
allocate(glob_u(allnodes))
allocate(lgcord(2,allnodes))
allocate(glob_cord(2,allnodes))
lg_u = 0.0
glob_u = 0.0
if (p .gt. 1) then
   do i = 1, nodes
      lg_u(localglobal(i)) = u(i)
   enddo
```

```
      call MPI_REDUCE(lg_u, glob_u, allnodes, MPI_REAL8, MPI_MAX, 0, &
                      MPI_COMM_WORLD, ierror)
   else
      glob_u = u
   endif
   do i = 1, elements
      do j = 1, 3
         elem(i,j) = localglobal(elem(i,j))
      enddo
   enddo
   if (p .gt. 1) then
      call MPI_ALLGATHER(elements, 1, MPI_INTEGER, vecelements, 1, &
                         MPI_INTEGER, MPI_COMM_WORLD, ierror)
      blen = 3*vecelements
      sum = 0
      do i = 1, p
         disp(i) = sum
         sum = sum + blen(i)
      enddo
      allelements = sum/3
   else
      allelements = elements
   endif
   allocate(glob_elem(3,allelements))
   allocate(elemadd(allelements))
   call TRANSPOSE_INT(elem, elements, 3, elemt)
   if (p .gt. 1) then
      call mpi_gatherv(elemt,3*elements, MPI_INTEGER, glob_elem, blen, disp, &
                       MPI_INTEGER, 0, MPI_COMM_WORLD, ierror)
      lgcord = 0.0
      do i = 1, nodes
         lgcord(1,localglobal(i)) = coord(i,1)
         lgcord(2,localglobal(i)) = coord(i,2)
      enddo
      call MPI_REDUCE(lgcord, glob_cord, 2*allnodes, MPI_INTEGER, &
                      MPI_MAX, 0, MPI_COMM_WORLD, ierror)
   else
      glob_cord(1,:) = coord(:,1)
      glob_cord(2,:) = coord(:,2)
      glob_elem = elemt
   endif
   if (rank .eq. 0) then
      print *, maxval(glob_u)
      do i = 1, gl_nbnodes
         gl_seg(i,1) = i
         gl_seg(i,2) = i + 1
      enddo
      gl_seg(gl_nbnodes,2) = 1
      segadd(:,1) = 0
      segadd(:,2) = 1
      segadd(:,3) = 1
      segadd(:,4) = 0
      segadd(:,5) = 1
      elemadd(1:allelements) = 1
      open(unit = 2, file = 'output.txt')
70    format (' ',I3)
      write(2,70) gl_nbnodes
```

```
      write(2,70) gl_seg(:,1)
      write(2,70) gl_seg(:,2)
      write(2,70) segadd(:,1)
      write(2,70) segadd(:,2)
      write(2,70) segadd(:,3)
      write(2,70) segadd(:,4)
      write(2,70) segadd(:,5)
      write(2,70) allelements
      write(2,70) glob_elem(1,1:allelements)
      write(2,70) glob_elem(2,1:allelements)
      write(2,70) glob_elem(3,1:allelements)
      write(2,70) elemadd(1:allelements)
      write(2,70) allnodes
80    format(' ',F9.4)
      write(2,80) glob_cord(1,1:allnodes)
      write(2,80) glob_cord(2,1:allnodes)
      write(2,80) glob_u(:)
      close(2)
   endif
   deallocate(elemadd)

   return
   end subroutine OUTPUT


   subroutine TRANSPOSE_INT(A, m, n, B)
   integer, intent(in):: m, n
   integer, intent(in):: A(m,n)
   integer, intent(out):: B(n,m)
   integer:: i, j

   do j = 1, n
      do i = 1, m
         B(j,i) = A(i,j)
      enddo
   enddo
   end subroutine TRANSPOSE_INT

   end module MESH_MODULE
```

B.3 sparse.f90

```
   module SPARSE_MODULE

   type sparsematrix
      integer:: length, rows
      real*8, pointer, dimension(:):: C
      integer, pointer, dimension(:):: J
      integer, pointer, dimension(:):: I
   endtype sparsematrix

   contains

! Initialize A with m rows:
   subroutine S_INIT(A, max, m)
```

```
   implicit none
   integer, intent(in):: max, m
   integer:: i
   type(sparsematrix), intent(out):: A

   allocate(A%C(max))
   allocate(A%J(max))
   allocate(A%I(m+1))
   A%length = max
   A%rows = m
   do i = 1, A%rows
      A%C(i) = 0.0
      A%J(i) = i
      A%I(i) = i
   enddo
   A%I(m+1) = m + 1
   end subroutine S_INIT

! Compute the matrix-vector product A*x::
   function S_MATVEC(A, x, n)
   implicit none
   type(sparsematrix), intent(in):: A
   real*8:: S_MATVEC(A%rows)
   integer, intent(in):: n
   real*8, intent(in):: x(n)
   real*8:: y(A%rows)
   integer:: i, k1, k2

   do i = 1, A%rows
      k1 = A%I(i)
      k2 = A%I(i+1) - 1
      y(i) = DOTPRODUCT(A%C(k1:k2),x(A%J(k1:k2)),k2-k1+1)
   enddo
   S_MATVEC = y
   end function S_MATVEC

! setup the stiffness matrix:
   subroutine STIFF_SETUP(A, dim, nv)
   implicit none
   integer, intent(in):: dim, nv
   type(sparsematrix), intent(out):: A
   integer:: i, index, rest

   call S_INIT(A, dim*6, dim)
   do i = 1, dim+1
      A%I(i) = 5*(i - 1) + 1
   enddo
   index = 0
   do i = 1, dim
      rest = MOD(i,nv)
      index = index + 1
      if (i .gt. nv) then
         A%J(index) = i - nv
         A%C(index) = -1.0
      else
         A%J(index) = 1
         A%C(index) = 0.0
```

```
      endif
      index = index + 1
      if (nv .ne. 1 .and. rest .ne. 1) then
         A%J(index) = i - 1
         A%C(index) = -1.0
      else
         A%J(index) = 1
         A%C(index) = 0.0
      endif
      index = index + 1
      A%J(index) = i
      A%C(index) = 4.0
      index = index + 1
      if (rest .ne. 0) then
         A%J(index) = i + 1
         A%C(index) = -1.0
      else
         A%J(index) = 1
         A%C(index) = 0.0
      endif
      index = index + 1
      if (i .le. dim-nv) then
         A%J(index) = i + nv
         A%C(index) = -1.0
      else
         A%J(index) = 1
         A%C(index) = 0.0
      endif
   enddo
   return
   end subroutine STIFF_SETUP

! apply the SSOR preconditioner:
   subroutine APPLY_SSOR(r, z, dim, nv, omega)
   implicit none
   integer, intent(in):: dim, nv
   real, intent(in):: r(dim), omega
   real, intent(out):: z(dim)
   integer:: i, rest
   real:: factor, y(dim)

   factor = 0.25*omega
   y = r
   do i = 1, dim
      rest = MOD(i,nv)
      if (i .gt. nv) then
         y(i) = y(i) + factor*y(i-nv)
      endif
      if (nv .ne. 1 .and. rest .ne. 1) then
         y(i) = y(i) + factor*y(i-1)
      endif
   enddo
   z = y
   do i = dim, 1, -1
      rest = MOD(i,nv)
      if (i .le. dim-nv) then
         z(i) = z(i) + omega*z(i+nv)
```

```
      endif
      if (rest .ne. 0) then
         z(i) = z(i) + omega*z(i+1)
      endif
      z(i) = 0.25*z(i)
   enddo
   z = omega*(2 - omega)*z
   return
   end subroutine APPLY_SSOR

! serial CG algorithm:
   subroutine CG(A, b, x0, x, maxit)
   implicit none
   real*8, parameter:: eps = 0.0001
   integer, intent(in):: maxit
   type(sparsematrix), intent(in):: A
   real*8, intent(in):: b(A%rows), x0(A%rows)
   real*8, intent(out):: x(A%rows)
   integer:: i, m
   real*8:: r(A%rows), q(A%rows), Aq(A%rows)
   real*8:: alpha, beta, rnorm, rnormold, initerr

   m = A%rows
   x = x0
   r = b - S_MATVEC(A, x0, m)
   initerr = maxval(abs(r))
   q = r
   rnormold = DOTPRODUCT(r, r, m)
   do i = 1, maxit
      Aq =  S_MATVEC(A, q, m)
      alpha = rnormold/DOTPRODUCT(Aq,q,m)
      x = x + alpha*q
      r = r - alpha*Aq
      if ((maxval(abs(r)) < eps) .and. (maxval(abs(r))/initerr < 0.01)) then
         goto 10
      endif
      rnorm = DOTPRODUCT(r, r, m)
      beta = rnorm/rnormold
      q = r + beta*q
      rnormold = rnorm
   enddo
10 return
   end subroutine CG

! serial PCG algorithm, takes the SSOR preconditioner:
   subroutine PCG(A, b, x0, x, maxit, nvert, omega)
   implicit none
   real*8, parameter:: eps = 0.0001
   integer, intent(in):: maxit, nvert
   type(sparsematrix), intent(in):: A
   real*8, intent(in):: omega, b(A%rows), x0(A%rows)
   real*8, intent(out):: x(A%rows)
   integer:: i, m
   real*8:: r(A%rows), q(A%rows), Aq(A%rows), z(A%rows)
   real*8:: alpha, beta, rnorm, rnormold, initerr

   m = A%rows
```

```
   x = x0
   r = b - S_MATVEC(A, x0, m)
   initerr = maxval(abs(r))
   call APPLY_SSOR(r, z, m, nvert, omega)
   q = z
   rnormold = DOTPRODUCT(r, z, m)
   do i = 1, maxit
      Aq =  S_MATVEC(A, q, m)
      alpha = rnormold/DOTPRODUCT(Aq,q,m)
      x = x + alpha*q
      r = r - alpha*Aq
      if ((maxval(abs(r)) < eps) .and. (maxval(abs(r))/initerr < 0.01)) then
         goto 10
      endif
      call APPLY_SSOR(r, z, m, nvert, omega)
      rnorm = DOTPRODUCT(r, z, m)
      beta = rnorm/rnormold
      q = z + beta*q
      rnormold = rnorm
   enddo
10 return
   end subroutine PCG

! Compute the dotproduct of two vectors:
   real*8 function DOTPRODUCT(u,v,dim)
   implicit none
   integer, intent(in):: dim
   real*8, intent(in):: u(dim), v(dim)
   integer:: k
   real*8:: dp

   dp = 0
   do k = 1, dim
      dp = dp + u(k)*v(k)
   enddo
   DOTPRODUCT = dp
   return
   end function DOTPRODUCT


   subroutine TRANSPOSE(A, m, n, B)
   integer, intent(in):: m, n
   real*8, intent(in):: A(m,n)
   real*8, intent(out):: B(n,m)
   integer:: i, j

    do j = 1, n
      do i = 1, m
         B(j,i) = A(i,j)
      enddo
   enddo
   end subroutine TRANSPOSE

   end module SPARSE_MODULE
```

B.4 matmesh.m

```
function matmesh

fid = fopen('output.txt','r');
segments = fscanf(fid,'%7d',1)
E = fscanf(fid,'%7d',[segments,7]);
elements = fscanf(fid,'%7d',1)
T = fscanf(fid,'%7d',[elements,4]);
nodes = fscanf(fid,'%7d',1)
P = fscanf(fid,'%f',[nodes,2]);
U = fscanf(fid,'%f',nodes);
st = fclose(fid);
E = E';
P = P';
T = T';
pdemesh(P,E,T);
pause
pdesurf(P,T,U);
```

BIBLIOGRAPHY

[Ad75] Adams, R.A. (1975). *Sobolev spaces.* Academic Press, New York.

[Bj89] Bjørstad, P.E. and Widlund, O.B. (1989). To overlap or not to overlap: A note on a domain decomposition method for elliptic problems. *SIAM J. Sci. Stat. Comput.*, **10**(5), 1053-1061.

[Ch92] Chan, T.F. and Goovaerts, D. (1992). On the relationship between overlapping and nonoverlapping domain decomposition methods. *SIAM J. Matrix Anal. Appl.*, **13**, 663-670.

[Co90] Conway, J.B. (1990). *A course in functional analysis.* Springer-Verlag, New York.

[Dr89] Dryja, M. and Widlund, O.B. (1989). Some domain decomposition algorithms for elliptic problems. In *Iterative methods for large linear systems*, L. Hayes and D. Kincaid, eds., Academic Press, San Diego, pp. 273-291.

[Dr92] Dryja, M. and Widlund, O.B. (1992). Additive Schwarz methods for elliptic finite element problems in three dimensions. In *Fifth international symposium on domain decomposition methods for partial differential equations*, D.E. Keyes et al., eds., SIAM, Philadelphia, pp. 3-18.

[Dr94a] Dryja, M. and Widlund, O.B. (1994). Domain decomposition algorithms with small overlap. *SIAM J. Sci. Comput.*, **15**(3), 604-620.

[Dr94b] Dryja, M. and Widlund, O.B. (1994). Some recent results on Schwarz type domain decomposition algorithms. In *Domain decomposition methods in science and engineering*, A. Quarteroni et al., eds., American Mathematical Society, Providence, R.I., pp. 53-61.

[Fo95] Folland, G.B. (1995). *Introduction to partial differential equations.* Princeton University Press, Princeton, N.J.

[Gi77] Gilbarg, D. and Trudinger, N.S. (1977). *Elliptic partial differential equations of second order.* Springer-Verlag, Berlin.

[Go96] Golub, G.H. and van Loan, C.F. (1996). *Matrix computations.* Johns Hopkins University Press, Baltimore.

[Gr94] Gropp, W., Lusk, E. and Skjellum, A. (1994). *Using MPI: Portable parallel programming with the message-passing interface.* MIT Press, Cambridge.

[Ji90] Jin, H. and Wiberg, N.E. (1990). Two-dimensional mesh generation, adaptive remeshing and refinement. *Int. J. for Numerical Methods in Engineering*, **29**, 1501-1526.

[Jo87] Johnson, C. (1987). *Numerical solution of partial differential equations by the finite element method.* Cambridge University Press, Cambridge.

[Ku77] Kufner, A., John, O. and Fucik, S. (1977). *Function spaces.* Noordhoff International Publishing, Leyden.

[Li88] Lions, P.L. (1988). On the Schwarz alternating method I. In *First international symposium on domain decomposition methods for partial differential equations*, R. Glowinski et al., eds., SIAM, Philadelphia, pp. 1-42.

[Li89] Lions, P.L. (1989). On the Schwarz alternating method II: Stochastic interpretation and order properties. In *Domain decomposition methods*, T.F. Chan et al., eds., SIAM, Philadelphia, pp. 47-70.

[Lo85] Lo, S.H. (1985). A new mesh generation scheme for arbitrary planar domains. *Int. J. for Numerical Methods in Engineering*, **21**, 1403-1426.

[Pe87] Peraire, J., Vahdati, M., Morgan, K. and Zienkiwicz, O.C. (1987). Adaptive remeshing for compressible flow computations. *Journal of Computational Physics*, **72**, 449-466.

[Po98] Pozrikidis, C. (1998). *Numerical computation in science and engineering.* Oxford University Press, Oxford.

[Qu94] Quarteroni, A. and Valli, A. (1994). *Numerical approximation of partial differential equations.* Springer-Verlag, Berlin.

[Qu99] Quarteroni, A. and Valli, A. (1999). *Domain decomposition methods for partial differential equations.* Oxford University Press, Oxford.

[Ru73] Rudin, W. (1973). *Functional analysis.* McGraw-Hill, New York.

[Sa96] Saad, Y. (1996). *Iterative methods for sparse linear systems.* PWS Publishing Company, Boston.

[Sc70] Schwarz, H.A. (1870). Ueber einen Grenzübergang durch alternirendes Verfahren. *Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich*, **15**, 272-286.

[Si91] Simon, H.D. (1991). Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, **2**(2/3).

[Sm96] Smith, B.F., Bjørstad, P.E. and Gropp, W.D. (1996). *Domain decomposition. Parallel multilevel methods for elliptic partial differential equations.* Cambridge University Press, Cambridge.

[Sn98] Snir, M., Otto, S., Huss-Lederman, S., Walker, D. and Dongarra, J. (1998). *MPI-The complete reference: Volume 1, the MPI core.* MIT Press, Cambridge.

[To96] Topping, B.H.V. and Khan, A.I. (1996). *Parallel finite element computations.* Saxe-Coburg Publications, Edinburgh.

[Wu98] Wu, X. and Li, W. (1998). Performance models for scalable cluster computing. *Journal of Systems Architecture*, **44**, 189-205.

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First, to my supervisor Dr. Michael W. Smiley for his guidance and support throughout this research and the writing of this thesis. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Wolfgang Kliemann, Dr. Glenn Luecke, Dr. Stephen Willson, and my major professor, Dr. Don Pigozzi.